pyusb-docs

リリース *0.0*

2020 年 05 月 02 日

目次:

第 1 章

ライセンス

以下の条件が満たされている場合に限り、変更の有無にかかわらず、ソースおよびバイナリ形式での再配布および使用が許可されます:

1. ソースコードの再配布では、上記の著作権表示、この条件のリスト、および以下の免責事項を保持する必要があります。

2. バイナリ形式で再配布する場合は、上記の著作権表示、この条件のリスト、および以下の免責事項を、配布物とともに提供されるドキュメントやその他の資料に複製する必要があります。

3. 特定の事前の書面による許可なしに、著作権所有者の名前もその貢献者の名前も、このソフトウェアから派生した製品を推奨または宣伝するために使用することはできません。

このソフトウェアは、著作権者および寄稿者によって「現状のまま」提供され、商品性および特定の目的に対する適合性の黙示の保証を含むがこれに限定されない、明示または黙示の保証は否認されます。いかなる場合でも、著作権者または寄稿者は、直接的、間接的、偶発的、特別、例示的、または結果的な損害（代替商品またはサービスの購入、使用、データ、または利益の損失を含みますが、これらに限定されません)、またはビジネスの中断、責任の理論にかかわらず、契約の責任、厳格な責任、または不法行為（過失またはその他を含む）にかかわらず、本ソフトウェアの使用を問わず、責任を負わないものとします。

第 2 章

PyUSB 1.0 - Python からの容易な USB アクセス

2.1 はじめに

PyUSB モジュールは、ホストマシンのユニバーサルシリアルバス (USB) システムへの Python からの容易なアクセスを提供します。

0.4 バージョンまでは PyUSB は libusb の薄いラッパーでした。 1.0 バージョンでは大幅に改善され、現在の PyUSB は豊富な API を持ち、バックエンドに対して中立で使いやすい Python USB モジュールです。

ほとんどの Python モジュールと同様に、PyUSB のドキュメントは Python docstring に基づいているため、pydoc などのツールで操作できます。

チュートリアルもご覧ください docs/tutorial.rst

PyUSB は Linux と Windows で開発およびテストされています。Python >= 2.4 かつ ctypes かつ 組み込みバックエンドの少なくとも 1 つ が走るすべてのプラットフォームで正常に動作するはずです。

PyUSB は libusb 1.0、libusb 0.1、OpenUSB をサポートしています。一部の特別な場合を除いて、ユーザーはそのことを気にする必要はありません。

PyUSB について質問がある場合は、SourceForge でホストされている PyUSB メーリングリストをご利用ください。 PyUSB website にメーリングリストの購読方法に関する説明があります。

2.2 インストール

PyUSB は pip でインストールします:

```
pip install pyusb
```

あなたのシステムでは libusb(1.0 または 0.1) または OpenUSB をが走っている必要があることに注意してください。 Windows ユーザーの場合、libusb 1.0 DLL は releases で提供されます (7z アーカイブを参照)。ibusb Web サイトで更新を確認してください (http://www.libusb.info)。

2.3 バグの報告及びパッチのご提供について

一部の人々は私の電子メールに直接パッチを送信し、バグを報告してきますが、 github から行うようにしてください。そうしないと名前を追跡して確認ファイルに入れるのに苦労します。

第 3 章

PyUSB 1.0 を使ってプログラミング

3.1 手前味噌ですが…

PyUSB 1.0 は、容易なる USB アクセスを可能にする Python ライブラリです。 PyUSB はいくつかの機能を提供します:

100% Python で書かれています: C で記述された 0.x バージョンとは異なり、1.0 バージョンは Python で記述されています。 これにより、C のバックグラウンドがない Python プログラマーは PyUSB がどのように機能するかをよりよく理解できます。

特定のプラットフォームに偏らず、中立です: 1.0 バージョンは、フロントエンド-バックエンド スキーム 実装です。 これにより、システム固有の実装の詳細と API が分離されています。 2 つの層の間の接着剤は `IBackend` インターフェースです。 PyUSB には、libusb 1.0、libusb 0.1、OpenUSB 用の組み込みバックエンドが付属しています。 必要に応じて、独自のバックエンドを作成することもできます。

ポータビリティ: PyUSB は、Python>=2.4 かつ ctypes かつ サポートされている組み込みバックエンドの少なくとも 1 つを備えた任意のプラットフォームで実行する必要があります。

最強に簡単: USB デバイスとの通信がこれまでになく簡単になりました。 USB は複雑なプロトコルですが、PyUSB にはほとんどの一般的な構成に適したデフォルトの構成があります。

アイソクロナス転送 (isochronous transfers) のサポート: 基礎となるバックエンドがアイソクロナス転送 (isochronous transfers) をサポートしている場合、PyUSB はアイソクロナス (isochronous transfers) 転送をサポートします。

PyUSB は USB プログラミングの苦痛を軽減しますが、このチュートリアルでは、最小限の USB プロトコルのバックグラウンドがあることを前提としています。 USB について何も知らない場合は、優れた Jan Axelson の著書 **USB Complete** をお勧めします。(訳注: USB Complete: The Developer's Guide (Complete Guides series) (English Edition) https://www.amazon.co.jp/dp/B00U58R0FA/ 日本語版もあるが版数が古そう)

3.2 もうお話は充分、さぁコーディングしましょう！

3.2.1 紳士録

まず、PyUSB モジュールの概要を説明します。 PyUSB モジュールは usb パッケージの下にあり、以下のモジュールがあります:

Content	Description
core	メイン USB モジュール。
util	ユーティリティ関数。
control	Standard control requests.
legacy	バージョン 0.x 系 互換レイヤ。
backend	組み込みのバックエンドを含むサブパッケージ。

たとえば、core モジュールをインポートするには、次のように入力します:

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

3.2.2 それじゃあ初めましょう

以下は、見つかった最初の OUT エンドポイントに 'test' 文字列を送信する単純際まわりないプログラムです:

```
import usb.core
import usb.util

# find our device
dev = usb.core.find(idVendor=0xfffe, idProduct=0x0001)

# was it found?
if dev is None:
    raise ValueError('Device not found')

# set the active configuration. With no arguments, the first
# configuration will be the active one
dev.set_configuration()

# get an endpoint instance
cfg = dev.get_active_configuration()
intf = cfg[(0,0)]

ep = usb.util.find_descriptor(
    intf,
```

(次のページに続く)

(前のページからの続き)

```
    # match the first OUT endpoint
    custom_match = \
    lambda e: \
        usb.util.endpoint_direction(e.bEndpointAddress) == \
        usb.util.ENDPOINT_OUT)

assert ep is not None

# write the data
ep.write('test')
```

最初の 2 行で PyUSB パッケージモジュールをインポートします。 usb.core がメインモジュールで、 usb.util はユーティリティ関数です。その次の命令はデバイスを検索し、見つかった場合はインスタンスオブジェクトを返します。 なければ None が返されます。その後、使用する構成を設定します。必要な構成を示す引数が指定されていないことに注意してください。 ご覧のとおり、多くの PyUSB 関数には、ほとんどの一般的なデバイスのデフォルトがあります。この場合、その構成セットは最初に見つかったものです。

次に、私たちが関心のあるエンドポイントを探します。 私たちは最初のインターフェース内で検索します。 エンドポイントを見つけたら、データをエンドポイントに送信します。

私たちがエンドポイントアドレスを事前に知っている場合は、デバイスオブジェクトから write 関数を呼び出すだけです。

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

ここでは、私たちはエンドポイントアドレス *1* に文字列 'test' を書き込みます。 これらすべての機能については、次節で詳しく説明します。

3.2.3 問題は何？

エラー発生時、PyUSB の全ての関数は 1 つの例外を発生させます。Python 標準例外 に加えて、PyUSB は USB 関連エラーの usb.core.USBError を定義しています。

あなたは PyUSB ログ機能を使用することもできます。 logging モジュールを使用します。 これを有効にするには、環境変数 PYUSB_DEBUG をレベル名 critical、 error、 warning、info、 debug のいずれかで定義します。

デフォルトでは、メッセージは sys.stderr に送信されます。 必要に応じて、 PYUSB_LOG_FILENAME 環境変数を定義することで、ログメッセージをファイルにリダイレクトできます。 その値が有効なファイルパスである場合、メッセージはそれに書き込まれ、そうでない場合は、 sys.stderr に送信されます。

3.2.4 あなたはそこにいますか？

core モジュールの find() 関数は、システムに接続されたデバイスを見つけて列挙するために使用されます。 たとえば、デバイスの vendor ID(ベンダー ID) が *0xfffe* 、 product ID(製品 ID) が *0x0001* であるとします。 それを探したい場合は、次のように進めます:

```
import usb.core

dev = usb.core.find(idVendor=0xfffe, idProduct=0x0001)
if dev is None:
    raise ValueError('Our device is not connected')
```

以上で、関数はデバイスを表す usb.core.Device オブジェクトを返します。 デバイスが見つからない場合、None を返します。 実際には、デバイス・デスクリプタ (Device Descriptor) の任意のフィールドを使用できます。 たとえば、システムに USB プリンターが接続されているかどうかを確認するにはどうすればよいでしょうか。 これはとても簡単です:

```
# actually this is not the whole history, keep reading
if usb.core.find(bDeviceClass=7) is None:
    raise ValueError('No printer found')
```

7 は、USB 仕様によるプリンタークラスのコードです。 ちょっと待てや！存在するすべてのプリンターを列挙したい場合はどうなりますか？問題ありません:

```
# this is not the whole history yet...
printers = usb.core.find(find_all=True, bDeviceClass=7)

# Python 2, Python 3, to be or not to be
import sys
sys.stdout.write('There are ' + len(printers) + ' in the system\n.')
```

何が起きたのでしょう。少々説明する時がきました。find には find_all というパラメータがあり、デフォルトは False です。 false[*1] の場合、 find は指定された基準に一致する最初に見つかったデバイスを返します (詳細はもうちょい後で)。 *true* の値を与えると、 find は代わりに基準に一致するすべてのデバイスのリストを返します。 以上です！簡単でしょ。

これで終わり？いいえ。私はあなたにすべての歴史を話したわけではありません。多くのデバイスは実際にはクラス情報をデバイス・デスクリプタ (Device Descriptor) ではなくインターフェイス・デスクリプタ (Interface Descriptor) に入れます。 したがって、システムに接続されているすべてのプリンターを本当に見つけるには、すべての構成、次にすべてのインターフェイスを渡り歩き、いずれかのインターフェイスの *bInterfaceClass* フィールドが 7 に等しいかどうかを確認する必要があります。 私のような プログラマー なら、もっと簡単な方法があるのか疑問に思うでしょう。 もちろん答えは YES。 まず、接続されているすべてのプリンターを見つけるための最

[*1] True または False（先頭大文字）とは、Python 言語のそういう値を意味します。そして私が true や false を言うとき、それは true と false に評価される Python のあらゆる表現を意味します。

終的なコードを見てみましょう。

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

class find_class(object):
    def __init__(self, class_):
        self._class = class_
    def __call__(self, device):
        # first, let's check the device
        if device.bDeviceClass == self._class:
            return True
        # ok, transverse all devices to find an
        # interface that matches our class
        for cfg in device:
            # find_descriptor: what's it?
            intf = usb.util.find_descriptor(
                                        cfg,
                                        bInterfaceClass=self._class
                                )
            if intf is not None:
                return True

        return False

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

custom_match パラメータは、デバイスオブジェクトを受け取る呼び出し可能なオブジェクトを受け入れます。一致するデバイスの場合は true を返し、一致しないデバイスの場合は false を返す必要があります。 必要に応じて、 custom_match とデバイスフィールドを組み合わせることもできます:

```
# find all printers that belongs to our vendor:
printers = usb.core.find(find_all=1, custom_match=find_class(7), idVendor=0xfffe)
```

今ここでは、*0xfffe* ベンダーのプリンターのみに関心があります。

3.2.5 お主、何者じゃ！

やった！デバイスを見つけました。が、そのデバイス君とお話をする前に、構成、インターフェース、エンドポイント、転送タイプなどについて詳しく知りたいと思います…

あなたがデバイスを持っている場合は、あなたはオブジェクトのプロパティとして任意のデバイス・デスクリプタ・フィールドにアクセスできます:

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

デバイスで有効な構成にアクセスするときは、あなたはデバイスを反復処理 (iterate) できます:

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

同様に、構成を反復 (iterate) してインターフェイスにアクセスしたり、インターフェイスを反復 (iterate) してエンドポイントにアクセスしたりできます。 各種類のオブジェクトには、それぞれのデスクリプタのフィールドが属性として含まれています。 例を見てみましょう:

```
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')
```

次のように、添え字を使用してデスクリプタにランダムにアクセスすることもできます:

```
>>> # access the second configuration
>>> cfg = dev[1]
>>> # access the first interface
>>> intf = cfg[(0,0)]
>>> # third endpoint
>>> ep = intf[2]
```

ご覧のとおり、インデックスはゼロベースです。 ちょっと待って！ 私がインターフェイスにアクセスする方法に奇妙な点があります…はい、そうです、構成の添え字は 2 つのアイテムのシーケンスを受け入れます。最初のアイテムはインターフェイスのインデックスで、2 番目のアイテムは代替設定です。したがって、最初のインターフェイスにアクセスするが、2 番目の代替設定には、 `cfg[(0,1)]` と書き込みます。

それでは、デスクリプタを見つける強力な方法である `find_descriptor` ユーティリティ関数について学びましょう。 私たちはプリンター検索の例でそれをすでに見ています。 `find_descriptor` は `find` とほぼ同じように動作しますが、2 つの例外があります:

- `find_descriptor` は、最初のパラメータとして、あなたが検索したいデスクリプタの親デスクリプタを

受け取ります。

- backend[*2] パラメータはありません。

たとえば、構成デスクリプタ cfg があり、インターフェース 1 のすべての代替設定を検索したい場合は、次のようにします:

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

find_descriptor は usb.util モジュールにあることに注意してください。 また、前述の custom_match パラメータも受け入れます。

同じデバイスを複数扱うには

コンピューター同じデバイスが 2 接続されていることがあります。それらをどのように区別すればよいでしょうか？ Device オブジェクトには、USB 仕様の一部ではないが、非常に便利な 2 つの追加属性があります。それは bus と address 属性です。ただし、これらの属性はバックエンドからのものであり、バックエンドがそれらをサポートしないことは自由です。その場合、それらは None に設定されます。そうじゃない場合は、これらの属性はデバイスのバス番号とバスアドレスを表し、あなたのご想像どおり、同じ idVendor と idProduct 属性を持つ 2 つのデバイスを区別するために使用できます。

3.2.6 私はどのようにすればいいですか？

接続された USB デバイスは、いくつかの標準的なリクエストを使って構成する必要があります。私は USB 仕様の調査を始めたとき、デスクリプタ (descriptors)、構成 (configurations)、インターフェイス (interfaces)、代替設定 (alternate settings)、転送タイプ (trannsfer types)、その他もろもろで混乱していました。そして最悪なことに、たったひとつの構成設定でも無視したら動かないのです。 PyUSB はあなたがこれをできるだけ簡単にできるようにすることを目指します。たとえば、デバイス・オブジェクトを取得した後、それと通信する前に最初に行う必要があることの 1 つは、 set_configuration リクエストを発行することです。このリクエストのパラメータは、あなたが関心のある構成 (configuration) にある bConfigurationValue です。 ほとんどのデバイスには構成 (configuration) が 1 つしかなく、使用する構成値を追跡するのは面倒です (私が見たほとんどは単純にハードコーディングしています)。それゆえ PyUSB では、あなたが引数なしで set_configuration を呼び出すだけで済むようにしてあります。 この場合、最初に見つかった構成 (configuration) が設定されます (デバイスに 1 つしかない場合は、その構成値を気にする必要はありません)。 たとえば、 *bConfigurationValue* フィールドが 5[*3] に等しい構成 (configuration) デスクリプタが 1 つあるデバイスがあるとします。以下の呼び出しはいずれも同様に機能します:

[*2] 各々のバックエンドのドキュメントを参照してください。

[*3] USB 仕様では、構成 (configuration) に順番になるような値を義務付けていません (訳注:次の値が +1 で得られると期待するな、飛び飛びバラバラを覚悟せよということ)。 インターフェイスと代替設定 (alternate setting) 番号についても同様です。

```
>>> dev.set_configuration(5)
# or
>>> dev.set_configuration() # we assume the configuration 5 is the first one
# or
>>> cfg = util.find_descriptor(dev, bConfigurationValue=5)
>>> cfg.set()
# or
>>> cfg = util.find_descriptor(dev, bConfigurationValue=5)
>>> dev.set_configuration(cfg)
```

えっへん！ Configuration オブジェクトを set_configuration のパラメータとして使用できます！ はい、そしてそれ自身を当座 (current) の構成 (configuration) として設定するための set メソッドもあります。

構成する必要がある場合と必要ない場合があるもう 1 つの設定は、インターフェイスの代替設定です。各デバイスは一度に 1 つのアクティブ化された構成 (configration) のみを持つことができ、各構成 (configuration) には複数のインターフェイスがあり、すべてのインターフェイスを同時に使用できます。 インターフェイスを論理デバイスと考えると、この概念をよりよく理解できます。 たとえば、多機能プリンターを想像してみましょう。これは同時にプリンターとスキャナーです。 物事をシンプルに (または少なくともできるだけシンプルに) 保つために、構成 (configuration) が 1 つしかないと考えてみましょう。 プリンターとスキャナーがあるため、構成には 2 つのインターフェースがあり、1 つはプリンター用、もう 1 つはスキャナー用です。 複数のインターフェースを持つデバイスは、複合デバイスと呼ばれます。 多機能プリンターをコンピューターに接続すると、オペレーティングシステムは 2 つの異なるドライバーをロードします。すなわち、あなたがお持ちの「論理」機器ごとに 1 つです。[*4]

じゃあ代替設定 (alternate setting) の方はどうなの？ ええ、インターフェイスには 1 つ以上の代替設定があります。 代替設定が 1 つしかないインターフェイスは、代替設定がないと見なされます ([*5])。 代替設定 (alternate setting) はインターフェイス用であり、構成 (configuration) はデバイス用です。つまり、各インターフェイス対して、アクティブにできる代替設定 (alternate setting) は 1 つだけです。 たとえば、USB 仕様では、デバイスのプライマリ代替設定 (alternate setting) にアイソクロナス・エンドポイント (isochronous endpoint) を含めることはできないため ([*6]) 、ストリーミングデバイスには少なくとも 2 つの代替設定 (alternate setting) が必要で、2 番目の設定にはアイソクロナス・エンドポイント (isochronous endpoint) があります。 ただし、構成 (configuration) とは異なり、代替設定 (alternate setting) が 1 つだけのインターフェイスを設定する必要はありません ([*7])。 set_interface_altsetting 関数でインターフェイスの代替設定 (alternate setting) を選択します:

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

警告: USB 仕様では、追加の代替設定 (alternate settings) のないインターフェイスに対する SET_INTERFACE リクエストを受信した場合、デバイスはエラーを返すことが許可されているとされています。したがって、インターフェースに複数の代替設定 (alternate settings) があるか、または SET_INTERFACE リクエストを受け

[*4] 実際はもう少し複雑ですが、ここでの説明はとしては十分です。
[*5] 私もそれが奇妙に聞こえることは知っています。
[*6] これは、デバイスの構成 (configuration) 時にアイソクロナス転送用の帯域幅がない場合、デバイスを成功裏に列挙できるためです。
[*7] デバイスは未構成の状態 (unconfigured state) であることが許可されているため、これは構成 (configration) では発生しません。

入れるかどうか不明な場合は、次のように try-except ブロック内で set_interface_altsetting を呼び出すのが最も安全な方法です:

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

あなたは関数のパラメータとして Interface オブジェクトを使用することもでき、 その際、 interface と alternate_setting パラメータは bInterfaceNumber と bAlternateSetting フィールドから自動的に推論されます。 例えば:

```
>>> intf = find_descriptor(...)
>>> dev.set_interface_altsetting(intf)
>>> intf.set_altsetting() # wow! Interface also has a method for it
```

警告: その Interface オブジェクトはアクティブな構成 (configuration) デスクリプタに属している必要があります。

3.2.7 私とお話して頂戴

いよいよ USB デバイスと通信する方法を学ぶ時が来ました。 USB には、バルク (bulk) と割り込み (interrupt) とアイソクロナス (isochronous) と制御 (control) の、4 種類の転送方法があります。 私は各転送方法の目的とそれらの間の違いを説明するつもりはありません。私は、あなたが少なくとも USB 転送の基本を知っているものとして扱います。

制御 (control) 転送は、仕様に記載されている構造化データを持つ唯一の転送です。他の転送は、USB の観点からは生データを送受信するだけです。 そのため、制御 (control) 転送を処理するためだけに別の関数があり、他のすべての転送は同じ関数によって管理されます。

あなたは ctrl_transfer メソッドを介して制御 (control) 転送を発行します。それは OUT 転送と IN 転送の両方に使用されます。転送方向は bmRequestType パラメータによって決定されます。

ctrl_transfer パラメータは制御要求構造 (control request structure) とほとんど同じです。 以下は、制御 (control) 転送を行う方法の例です ([*8]):

```
>>> msg = 'test'
>>> assert dev.ctrl_transfer(0x40, CTRL_LOOPBACK_WRITE, 0, 0, msg) == len(msg)
```

(次のページに続く)

[*8] PyUSB では、制御 (control) 転送はエンドポイント 0 でのみ発行されます。代替制御エンドポイント (alternate control endpoint) を持つデバイスは非常に非常にまれです（私はそのようなデバイスを全く見たことがありません)。

(前のページからの続き)

```
>>> ret = dev.ctrl_transfer(0xC0, CTRL_LOOPBACK_READ, 0, 0, len(msg))
>>> sret = ''.join([chr(x) for x in ret])
>>> assert sret == msg
```

この例では、デバイスがループバック・パイプとして機能する 2 つのカスタム制御要求 (custom control request) を実装していると想定しています。 CTRL_LOOPBACK_WRITE メッセージで書き込んだものは、CTRL_LOOPBACK_READ メッセージで読み取ることができます。

最初の 4 つのパラメータは、標準制御転送構造 (standard control transfer) のフィールド bmRequestType、bmRequest、 wValue、 wIndex です。 5 番目のパラメーター (訳注:data_or_wLength=None) は、OUT 転送のデータ・ペイロード、または IN 転送で読み取るバイト数です。 データペイロードは、array __init__ メソッドのパラメーターとして使用できる任意のシーケンス型にすることができます。 データ・ペイロードがない場合、パラメータは None (または IN 転送の場合は 0) でなければなりません。 操作のタイムアウトを指定する最後のオプションパラメータ (訳注:timeout=None) が 1 つあります。 指定しない場合は、デフォルトのタイムアウトが使用されます (詳しくは後で説明します)。 OUT 転送では、戻り値は実際にデバイスに送信されたバイト数です。IN 転送では、戻り値はデータが読み込まれた array オブジェクトです。

他の転送については、メソッド write と read を使用して、データの書き込みと読み取りを行います。転送タイプを気にする必要はありません。転送タイプはエンドポイント・アドレスから自動的に決定されます。エンドポイント 1 にループバック・パイプがあると想定したループバックの例を以下に示します:

```
>>> 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
```

最初 (endpoint) と 3 番目 (timeout) のパラメーターは read/write 両方のメソッドで等しくそれぞれエンドポイント・アドレスとタイムアウトです。 2 番目のパラメーターは、データ・ペイロード (write) または読み取るバイト数 (read) です。 read メソッドの戻り値は array オブジェクトのインスタンスで、write メソッドの戻り値はで書き込んだバイト数です。

ベータ 2 バージョン以降、バイト数の代わりに、データが読み込まれる array オブジェクトを read と ctrl_transfer に渡すこともできます。 この場合、読み込むバイト数は、配列の長さと array.itemsize 値の積になります。

ctrl_transfer と同様に、 timeout パラメータはオプションです。 timeout が省略された場合、操作のタイムアウト値として Device.default_timeout プロパティが使用されます。

3.2.8 自分自身を制御する

転送関数に加えて、モジュール usb.control は、標準の USB 制御要求 (standard USB control request) を実装する関数を提供し、 usb.util モジュールには、文字列デスクリプタを返すための便利な関数 get_string があ

ります。

3.3 オマケ

3.3.1 優れた抽象化の背後には、優れた実装があります

初期の頃は libusb しかありませんでした。次に libusb 1.0 が登場し、libusb 0.1 と 1.0 がリリースされました。 その後、OpenUSB が開発され、現在私たちは、USB ライブラリのバベルの塔 (Tower of Babel) に住んでいます ([*9])。 PyUSB はそれをどのように扱いますか？ ええ、PyUSB は民主的なライブラリです。あなたは好きなライブラリを選択できます。 実際、独自の USB ライブラリを最初から作成して、PyUSB に使用するように指示することができます。 しかし、あなたはたぶん libusb 1.0 を使用するのがいいでしょう。

`find` 関数には、まだ説明していないパラメータがあります。これは `backend` パラメータです。 指定しない場合は、組み込みのバックエンドの 1 つが使用されます。 バックエンドは `usb.backend.IBackend` から継承されたオブジェクトで、オペレーティング・システム固有の USB の実装を担当します。ご想像のとおり、既に PyUSB に組み込み済なのは libusb 1.0(デフォルト) と libusb 0.1 と OpenUSB(非推奨) バックエンドです。

あなたは独自のバックエンドを作成・使用することができます。 `IBackend` から継承し、必要なメソッドを実装するだけです。 方法については、 `usb.backend` パッケージのドキュメントをご覧ください。

3.3.2 ホンマわがままやなぁ…

Python には、**自動メモリ管理** (automatic memory management) と呼ばれるものがあります。 これは、仮想マシンがオブジェクトをメモリから解放するタイミングを決定することを意味します。 内部的には、PyUSB は動作する必要のあるすべての低レベルのリソース (インターフェイスの要求、デバイス・ハンドルなど) を管理し、ほとんどのユーザーはそのことを心配する必要はありません。ただし、Python のオブジェクトの自動破棄には非決定的 (nondetrministic) な性質があるため、ユーザーは割り当てられたリソースがいつ解放されるかを予測できません。一部のアプリケーションは、確定的にリソースを割り当てて解放する必要があります。これらの種類のアプリケーションのために、 `usb.util` モジュールはリソース管理を扱うための一連の関数を持っています。

インターフェイス (interface) を手動で要求 (claim) および解放 (release) したい場合は、 `claim_interface` 関数と `release_interface` 関数を使用できます。 `claim_interface` は、デバイスが、あなたが指定したインターフェイスをまだ要求していない場合、それを要求します。 デバイスが既にインターフェイスを要求済である場合は何もしません。 同様に、 `release_interface` は、あなたが指定のインターフェイスが既に要求 (claim) 済の場合、それを解放します。まだインターフェイスが要求されていない場合は何もしません。この手動インターフェイス要求 (manual interface claim) を使用して、libusb のドキュメントに記載されている構成選択の問題 (configuration selection problem) を解決できます。

あなたが、デバイス・オブジェクトによって割り当てられたすべてのリソース (要求済のインターフェースを含む)

[*9] 単なる冗談です。真剣に受け止めないでください。 でも、多くの選択肢は、選択肢がないよりも優れています。

を解放したい場合は、 dispose_resources 関数を使用できます。 割り当てられたすべてのリソースを解放し、デバイス・オブジェクト (デバイス・ハードウェア自体ではない) を、find 関数呼び出しから戻ってきた直後の状態にします。

3.3.3 ライブラリを手動で指定する

一般に、バックエンドは、USB アクセス API を実装する共有ライブラリのラッパーです。 デフォルトでは、バックエンドは find_library() ctypes 関数を使用します。 Linux や他の UNIX 風オペレーティングシステムでは、find_library は外部プログラム (*/sbin/ldconfig* や *gcc* や *objdump* など) を実行してライブラリファイルを見つけようとします。

これらのプログラムがないか、ライブラリ・キャッシュが無効になっているシステムでは、この機能は使用できません。 この制限を克服するために、PyUSB ではカスタムの *find_library()* 関数をバックエンドに提供できます。

このようなシナリオの例です:

```
>>> 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)
```

get_backend() 関数の *find_library* 引数に注意してください。この引数では、バックエンドの正しいライブラリを見つける責任がある関数を指定します。

3.3.4 古臭いルール

あなたが古い PyUSB API(バージョン 0.X 系統) を使用してアプリケーションを作成していた場合、新しい API を使用するようにコードを更新する必要があるかどうかを悩んでいるかもしれません。まあ、できればそうすべきですが、そうする必要はありません。 PyUSB 1.0 には usb.legacy 互換モジュールが付属しています。新しい API の上に古い API を実装します。「では、アプリケーションを機能させるために、import usb を import usb.legacy as usb に置き換えるだけでよいですか？」答えは「はい」です。これでうまくいきますが、でも、これすらも必要はありません。 アプリケーションをそのまま実行すると、 usb.legacy からすべてのパブリック・シンボルが import usb ステートメントによってインポートされるため、そのまま機能します。問題が発生した場合は、おそらくあなたは何かしらバグを発見したということです。(訳注:とは言え、互換なのはあくまで PyUSB だけであって PyUSB が参照する Python 標準ライブラリが変更になっていることがある。手元では USBError.message を USBError.strerror に修正する必要があった。USBError が IOError を継承しているため。)

第 4 章

FAQ

4.1 "No backend available" エラーを修正するにはどうすればいいですか?

通常、この問題には次の 4 つの原因が考えられます:

1. あなたが libusb ライブラリをインストールしていない場合。
2. あなたが libusb ライブラリを標準の共有ライブラリパス置いてない場合。
3. あなたの libusb のバージョンがとても古い場合。
4. あなたの PyUSB バージョンがとても古い場合。

何が問題なのかをデバッグするには、あなたの環境で以下のスクリプトを実行します:

```
import os
os.environ['PYUSB_DEBUG'] = 'debug'
import usb.core
usb.core.find()
```

これにより、デバッグメッセージがコンソールに出力されます。 それでも何が起こっているのか分からないときは、当該のデバッグ出力を掲載して、メーリングリストに助けを求めてください。

4.2 Windows に libusb をインストールするにはどうすればいいですか?

libusb または libusb-win32 を Windows にインストールするには、 zadig を使用してください。

4.3 指定のバックエンドの使用を強制するにはどうすればいいですか？

libusb1 バックエンド使用例:

```
>>> import usb.core
>>> from usb.backend import libusb1
>>> be = libusb1.get_backend()
>>> dev = usb.core.find(backend=be)
```

4.4 libusb ライブラリ・パスをバックエンドに渡すにはどうすればよいですか？

tutorial の *Specify libraries by hand* (邦訳:ライブラリを手動で指定する) を参照。

4.5 選択した構成 (configuration) の、すでに構成 (configured) されているデバイスで set_configuration() を 呼び出す/呼び出さない 方法は？

通常、 `set_configuration()` はデバイスの初期化中に呼び出されます。 libusb documentation の `libusb_set_configuration()` は次のように述べています:

> あなたが、選択した構成 (configuration) で既に構成 (configure) されているデバイスでこの関数を呼び出すと、この関数は軽量のデバイスリセット (lightweight device reset) として機能します。すなわち、当座 (current) の構成 (configuration) を使用して SET_CONFIGURATION リクエストを発行し、ほとんどの USB 関連デバイスの状態をリセットします (altsetting をゼロにリセットし、エンドポイントをクリアし停止、reset をトグル)。

したがって、その後 `write()` を呼び出すと、タイムアウトエラーが発生します。

この動作に対する解決策の一つは、 configuration selection and handling (構成 (configuration) の選択と処理) で説明されているように、現在アクティブな構成を検討することです。 "必要な構成 (configuration) が既にアクティブである場合、構成 (configuration) を選択する必要はありません"

```
try:
    cfg = dev.get_active_configuration()
except usb.core.USBError:
    cfg = None
if cfg is None or cfg.bConfigurationValue != cfg_desired:
    dev.set_configuration(cfg_desired)
```

第 5 章

usb パッケージ

5.1 サブパッケージ

5.1.1 usb.backend パッケージ

サブモジュール

usb.backend.libusb0 モジュール

usb.backend.libusb0.**get_backend**(*find_library=None*)

usb.backend.libusb1 モジュール

usb.backend.libusb1.**get_backend**(*find_library=None*)

usb.backend.openusb モジュール

モジュール内容

usb.backend - バックエンド・インターフェイス

本モジュールからは以下がエスクポートされます:

IBackend - バックエンド・インターフェイス

バックエンド (backend) は、IBackend インターフェイスを実装する Python オブジェクトです。 これを行う最も簡単な方法は、IBackend から継承することです。

PyUSB はすでに libusb バージョン 0.1 と 1.0、および OpenUSB ライブラリのバックエンドを提供しています。PyUSB に含まれるバックエンド・モジュールは、バックエンド・オブジェクトのインスタンスを返す get_backend() 関数をエクスポートする必要があります。必要に応じて、独自にカスタマイズしたバックエンドを提供できます。以下に、バックエンド実装モジュールのスケルトンを示します:

import usb.backend

class MyBackend(usb.backend.IBackend): pass

def get_backend(): return MyBackend()

あなたは、あなたがカスタマイズしたバックエンドを usb.core.find() 関数のバックエンド・パラメータとして渡すことで使用できます。例えば:

import custom_backend import usb.core

myidVendor = 0xfffe myidProduct = 0x0001

mybackend = custom_backend.get_backend()

dev = usb.core.find(backend = mybackend, idProduct=myidProduct, idVendor=myidVendor)

カスタム・バックエンドの場合、アプリケーション・コードがバックエンドをインスタンス化するため、あなたは get_backend() 関数を提供する必要はありません。

find() 関数にバックエンドを指定しない場合、内部ルールに従ってデフォルトのバックエンドの 1 つを使用します。詳細については、find() 関数のドキュメントを参照してください。

class usb.backend.**IBackend**

ベースクラス: usb._objfinalizer.AutoFinalizedObject

バックエンド・インターフェイス

IBackend は、バックエンド実装の基本的なインターフェイスです。 デフォルトでは、インターフェイスのメソッドは NotImplementedError 例外を発生させます。 バックエンド実装は、必要な機能を提供するためにメソッドを置き換える必要があります。

Python は動的型付き言語であるため、IBackend から継承する義務はありません。IBackend のように動作するものはすべて IBackend です。 ただし、IBackend から継承することをことを強くお勧めします。IBackend から継承すると、一貫したデフォルトの動作が提供されます。

attach_kernel_driver(*dev_handle*, *intf*)

以前 detach_kernel_driver() を使用して切り離されていたインターフェイスのカーネル・ドライバーを再接続します。

bulk_read(*dev_handle*, *ep*, *intf*, *buff*, *timeout*)

一括読み取り (bulk read) を実行します。

dev_handle は、 open_device() メソッドによって返される値です。 ep パラメーターは、データの受信元となるエンドポイントの bEndpointAddress フィールドです。 intf は、エンドポイントを含むインターフェイスの bInterfaceNumber フィールドです。 buff パラメータは、読み取られたデータを受け取るためのバッファです。バッファの長さのバイト数だけデータが読み取られます。timeout パラメータは、操作のタイムアウトをミリ秒単位で指定します。

本メソッドは実際に読み取ったバイト数を返します。

bulk_write(*dev_handle*, *ep*, *intf*, *data*, *timeout*)
一括書き込み (bulk write) を実行します。

dev_handle は、open_device() メソッドによって返される値です。 ep パラメータは、データが送信されるエンドポイントの bEndpointAddress フィールドです。 intf は、エンドポイントを含むインターフェイスの bInterfaceNumber フィールドです。 data パラメータは、送信されるデータです。 これは、array.array クラスのインスタンスでなければなりません。 timeout パラメータは、操作のタイムアウトをミリ秒単位で指定します。

本メソッドは書き込んだバイト数を返します。

claim_interface(*dev_handle*, *intf*)
指定のインターフェイスを要求 (claim) します。

インターフェイスの要求 (claim) は USB 仕様自体には関係ありませんが、通常は USB ライブラリに必要な呼び出しです。システム上のインターフェイスへの排他的アクセスを要求 (request) します。このメソッドは、いずれかの転送 (transfer) メソッドを使用する前に呼び出す必要があります。

dev_handle は open_device() メソッドによって返された値で、intf は欲しいインターフェイスの bInterfaceNumber フィールドです。

clear_halt(*dev_handle*, *ep*)
エンドポイントの停止/失速 (halt/stall) 状態をクリアします。

close_device(*dev_handle*)
デバイス・ハンドルをクローズする。

このメソッドは、デバイスの通信チャネルを閉じ、それに関連するすべてのシステムリソースを解放します。

ctrl_transfer(*dev_handle*, *bmRequestType*, *bRequest*, *wValue*, *wIndex*, *data*, *timeout*)
エンドポイント 0(ゼロ) で制御 (control) 転送を実行します。

転送の方向は、セットアップパケットの bmRequestType フィールドから推測されます。

dev_handle は、 open_device() メソッドによって返される値です。 bmRequestType と bRequest と wValue と wIndex は、セットアップパケットの同名のフィールドです。 data は配列オブジェクトです。OUT リクエストの場合はデータ・ステージで送信するバイトを格納しており、IN リクエストの場合は読み取ったデータを保持するバッファです。 送信または受信が要求されたバイト数は、配列の長さに data.itemsize フィールドを掛けたものと同じです。 timeout パラメータは、操作のタイムアウトをミリ秒単位で指定します。

OUT 転送時は書き込んだバイト数を返します。IN 転送時は読み込んだデータを array.array オブジェクトとして返します。

detach_kernel_driver(*dev_handle*, *intf*)
インターフェイスからカーネル・ドライバを取り外します。

成功ならば、あなたはインターフェイスを要求 (claim) して入出力を実行可能です。

enumerate_devices()
この関数は、システムで検出された各 USB デバイスの実装定義のデバイス ID を生成する反復可能 (iterable) なオブジェクトを返すために必要です。

デバイス識別オブジェクト (device identification object) は、インターフェイスの他のメソッドへの引数として使用されます。

get_configuration(*dev_handle*)
当座 (current) のアクティブなデバイス構成 (configuration) を取得。

このメソッドは、現在アクティブな構成 (configuration) の bConfigurationValue を返します。バックエンドと OS に応じて、キャッシュされた値が返されるか、制御要求 (control request) が発行されます。dev_handle パラメータは、open_device メソッドによって返される値です。

get_configuration_descriptor(*dev*, *config*)
与えたデバイスの構成 (configuration) デスクリプタを返します。

返されるオブジェクトには、メンバー変数としてアクセス可能なすべての構成 (configuration) デスクリプタ・フィールドが必要です。これらは、int 型に変換可能である必要があります (等しい必要はありません)。

dev パラメータはデバイス識別オブジェクトです。config は構成 (congigure) の論理インデックスです (bConfigurationValue フィールドではありません)。「論理インデックス」とは、GET_DESCRIPTOR リクエストの結果としてペリフェラルによって返される構成の相対的な順序を意味します。

get_device_descriptor(*dev*)
与えたデバイスのデバイス・デスクリプタを返します。

返されるオブジェクトには、すべてのデバイス・デスクリプタ・フィールドをメンバー変数としてアクセス可能にする必要があります。 これらは、int 型に変換可能である必要があります (等しい必要はありません)。

dev は、enumerate_devices() メソッドによって返されるイテレータによって生成されるオブジェクトです。

get_endpoint_descriptor(*dev*, *ep*, *intf*, *alt*, *config*)
与えたデバイスのエンドポイント・デスクリプタを返します。

返されるオブジェクトには、メンバー変数としてアクセス可能なすべてのエンドポイント・デスクリプタ・フィールドが必要です。 これらは、int 型に変換可能である必要があります (等しい必要はありません)。

ep パラメーターは、必要なエンドポイント・デスクリプタのエンドポイント論理インデックス (bEndpointAddress フィールドではない) です。 dev、intf、alt、config は、get_interface_descriptor() メソッドですでに説明済のものと同じ値です。

get_interface_descriptor(*dev*, *intf*, *alt*, *config*)
与えたデバイスのインターフェイス・デスクリプタを返します。

返されるオブジェクトには、すべてのインターフェイス・デスクリプタ・フィールドをメンバー変数としてアクセスできるようにする必要があります。 これらは、int 型に変換可能である必要があります (等しい必要はありません)。

dev パラメータはデバイス識別オブジェクトです。 intf パラメータは (bInterfaceNumber フィールドではなく) インターフェイスの論理インデックスであり、alt は (bAlternateSetting 値ではなく) 代替設定 (alternate setting) の論理インデックスです。すべてのインターフェイスに複数の代替設定 (alternate setting) があるわけではありません。この場合、alt パラメータはゼロにする必要があります。config は構成 (configuration) の論理インデックスです (bConfigurationValue フィールドではありません)。

get_parent(*dev*)
与えたデバイスの親デバイスを返します。

intr_read(*dev_handle*, *ep*, *intf*, *size*, *timeout*)
割り込み読み取り (interrupt read) を実行します。

dev_handle は、open_device() メソッドによって返される値です。ep パラメーターは、データの受信元となるエンドポイントの bEndpointAddress フィールドです。intf は、エンドポイントを含むインターフェイスの bInterfaceNumber フィールドです。buff パラメータは、読み取られたデータを受け取るためのバッファです。バッファの長さのバイト数だけデータが読み取られます。timeout パラメータは、操作のタイムアウトをミリ秒単位で指定します。

本メソッドは実際に読み取ったバイト数を返します。

intr_write(*dev_handle*, *ep*, *intf*, *data*, *timeout*)
割り込み書き込み (interrupt write) を実行します。

dev_handle は、open_device() メソッドによって返される値です。 ep パラメータは、データが送信されるエンドポイントの bEndpointAddress フィールドです。 intf は、エンドポイントを含むインターフェイスの bInterfaceNumber フィールドです。 data パラメータは、送信されるデータです。 これは、array.array クラスのインスタンスでなければなりません。 timeout パラメータは、操作のタイムアウトをミリ秒単位で指定します。

本メソッドは書き込んだバイト数を返します。

is_kernel_driver_active(*dev_handle*, *intf*)
カーネル・ドライバーがインターフェイスでアクティブかどうかを確認します。

カーネル・ドライバがアクティブな場合、あなたはインターフェイスを要求 (claim) できず、バックエンドは入出力を実行できません。

iso_read(*dev_handle*, *ep*, *intf*, *size*, *timeout*)
アイソクロナス読み取りを実行します。

dev_handle は、open_device() メソッドによって返される値です。ep パラメーターは、データの受信元となるエンドポイントの bEndpointAddress フィールドです。intf は、エンドポイントを含むインターフェイスの bInterfaceNumber フィールドです。buff パラメータは、読み込まれたデータを受け取るためのバッファであり、バッファの長さのバイト数だけデータが読み込まれます。timeout パラメータは、操作のタイムアウトをミリ秒単位で指定します。

本メソッドは実際に読み取ったバイト数を返します。

iso_write(*dev_handle*, *ep*, *intf*, *data*, *timeout*)
アイソクロナス書き込みを実行します。

dev_handle は、open_device() メソッドによって返される値です。 ep パラメータは、データが送信されるエンドポイントの bEndpointAddress フィールドです。 intf は、エンドポイントを含むインターフェイスの bInterfaceNumber フィールドです。 data パラメータは、送信されるデータです。 これは、array.array クラスのインスタンスでなければなりません。 timeout パラメータは、操作のタイムアウトをミリ秒単位で指定します。

本メソッドは書き込んだバイト数を返します。

open_device(*dev*)
データ交換 (data exchange) のためにデバイスを開きます。

このメソッドは、通信のために dev パラメーターで識別されたデバイスを開きます。 このメソッドは、転送メソッドなどの通信関連メソッドを呼び出す前に呼び出す必要があります。

通信インスタンス (communication instance) を識別するハンドルを返します。 このハンドルを通信メソッド (communication method) に渡す必要があります。

release_interface(*dev_handle*, *intf*)
要求済のインターフェイス (claimed interface) を開放する。

dev_handle と intf は、claim_interface メソッドと同様のパラメーターです。

reset_device(*dev_handle*)
デバイスをリセットする。

set_configuration(*dev_handle*, *config_value*)
アクティブなデバイス構成 (configurtion) をセットする。

このメソッドは、デバイスのアクティブな構成を設定するために呼び出す必要があります。 dev_handle パラメータは open_device() メソッドによって返される値であり、config_value パラメータは関連する構成 (configure) デスクリプタの bConfigurationValue フィールドです。

set_interface_altsetting(*dev_handle*, *intf*, *altsetting*)
インターフェイス代替設定 (alternate setting) をセットする。

このメソッドは、インターフェイスに複数の代替設定 (alternate setting) がある場合にのみ呼び出す必要があります。dev_handle は、open_device() メソッドによって返される値です。intf と altsetting は、それぞれ関連するインターフェイスの bInterfaceNumber および bAlternateSetting フィールドです。

5.2 サブモジュール

5.3 usb.control モジュール

usb.control - USB 標準制御要求 (standard control requests)

本モジュールからは以下がエスクポートされます:

get_status - 受信者 (recipeint) ステータスの取得、 clear_feature - 受信者 (recipient) 機能 (feature) のクリア、set_feature - 受信者 (recipient) 機能 (feature) の設定、get_descriptor - デバイス・デスクリプタ取得、set_descriptor - デバイス・デスクリプタ設定、 get_configuration - デバイス構成 (configuration) の取得、 set_configuration - デバイス構成 (configuration) の設定 get_interface - デバイス・インターフェイス取得、 set_interface - デバイス・インターフェイス設定

usb.control.**get_status**(*dev*, *recipient=None*)

指定の受信者 (recipient) のステータスを返す。

dev は、リクエストの送信先となるデバイス・オブジェクトです。

受信者 (recipient) は、 None (デバイスからステータスが照会される) または Interface または Endpoint デスクリプタにすることができます。

ステータス値は整数として返され、下位ワードは 2 バイトのステータス値からなります。

usb.control.**clear_feature**(*dev*, *feature*, *recipient=None*)

指定の機能 (feature) をクリアまたは無効にします。

dev は、リクエストの送信先となるデバイス・オブジェクトです。

feature には、あなたが無効にしたい機能 (feature) を指定します。

受信者 (recipient) は、 None (デバイスからステータスが照会される) または Interface または Endpoint デスクリプタにすることができます。

usb.control.**set_feature**(*dev*, *feature*, *recipient=None*)

指定の機能 (feature) を設定または有効化します。

dev は、リクエストの送信先となるデバイス・オブジェクトです。

feature にはあなたが有効にしたい機能 (feature) を指定します。

受信者 (recipient) は、 None (デバイスからステータスが照会される) または Interface または Endpoint デスクリプタにすることができます。

usb.control.**get_descriptor**(*dev*, *desc_size*, *desc_type*, *desc_index*, *wIndex=0*)
指定のデスクリプタを返す。

dev は、リクエストの送信先となるデバイス・オブジェクトです。

desc_size はデスクリプタのサイズを指定します。

desc_type と desc_index は、それぞれデスクリプタのタイプとインデックスです。 wIndex インデックスは文字列デスクリプタに使用され、言語 ID を表します。文字列デスクリプタ以外タイプの記述子の場合、wIndex はゼロです。

usb.control.**set_descriptor**(*dev*, *desc*, *desc_type*, *desc_index*, *wIndex=None*)
存在するデスクリプタを更新するか、または新しいデスクリプタを追加します。

dev は、リクエストの送信先となるデバイス・オブジェクトです。

desc ほげほげなパラメータ達は、デバイスに送信されるデスクリプタです。 desc_type と desc_index は、それぞれデスクリプタのタイプとインデックスです。 wIndex インデックスは文字列デスクリプタに使用され、言語 ID を表します。 文字列デスクリプタ以外のタイプのデスクリプタの場合、wIndex はゼロです。

usb.control.**get_configuration**(*dev*)
デバイスの、当座 (current) でアクティブな構成 (configuration) を取得します。

dev は、リクエストの送信先となるデバイス・オブジェクトです。

キャッシュされたデータを使用する可能性がある Device.get_active_configuration メソッドとは異なり、この関数は常にデバイス・リクエストを実行します。

usb.control.**set_configuration**(*dev*, *bConfigurationNumber*)
当座 (current) とする構成 (configuration) を設定します。

dev は、リクエストの送信先となるデバイス・オブジェクトです。

usb.control.**get_interface**(*dev*, *bInterfaceNumber*)
インターフェイスの代替設定 (alternate setting) の当座 (current) を取得します。

dev は、リクエストの送信先となるデバイス・オブジェクトです。

usb.control.**set_interface**(*dev*, *bInterfaceNumber*, *bAlternateSetting*)
インターフェイスの代替設定 (alternate setting) を設定します。

dev は、リクエストの送信先となるデバイス・オブジェクトです。

5.4 usb.core モジュール

usb.core - コア USB 機能

本モジュールからは以下がエスクポートされます:

Device - USB デバイスを表すクラス、 Configuration - 構成 (configuration) デスクリプタを表すクラス、 Interface - インターフェイス・デスクリプタを表すクラス、 Endpoint - エンドポイント・デスクリプタを表すクラス、 find() - USB デバイスを探す関数、show_devices() - 存在するデバイスを表示する関数

class usb.core.**Device**(*dev*, *backend*)

ベースクラス: usb._objfinalizer.AutoFinalizedObject

Device オブジェクトです。

このクラスには、USB 仕様に基づくデバイス・デスクリプタのすべてのフィールドが含まれています。 あなたはそれらにクラスのプロパティとしてアクセスできます。 たとえば、デバイス・デスクリプタのフィールド bDescriptorType にアクセスするには、次のようにします:

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

さらに、クラスはハードウェアと通信するメソッドを提供します。 通常、アプリケーションは最初に set_configuration() メソッドを呼び出してデバイスを既知の構成済み状態 (a known conifgured state) にし、代替設定が複数ある場合、オプションで set_interface_altsetting() を呼び出して、使用するインターフェイスの代替設定を選択し、 write() や read() メソッドでデータを送受信します。

新しいハードウェアで作業する場合、最初の試行は次のようになります:

```
>>> import usb.core
>>> dev = usb.core.find(idVendor=myVendorId, idProduct=myProductId)
>>> dev.set_configuration()
>>> dev.write(1, 'test')
```

このサンプルでは、対象のデバイスを見つけ（myVendorId および myProductId をデバイスの対応する値で置き換える必要があります)、デバイスを構成し（デフォルトでは、構成値 (configuration value) は 1 であり、ほとんどのデバイスの一般的な値です)、エンドポイント 0x01 へデータを幾つか書き込みます。

書き込み、読み取り、および ctrl_transfer メソッドのタイムアウト値はミリ秒単位で指定します。このパラメーターを省略した場合、代わりに Device.default_timeout 値が使用されます。Device.default_timeout プロパティは、ユーザがいつでも設定できます。

attach_kernel_driver(*interface*)

以前に detach_kernel_driver() を使って切り離し (detach) した、指定のインターフェイスのカーネル・ドライバを再接続 (re-attach) します。

interface パラメータは、ドライバを接続したい、デバイスのインターフェイス番号を指定します。

backend
そのデバイスが使っているバックエンドを返します。

clear_halt(*ep*)
ep に指定したエンドポイントの halt/stall 状態をクリアします。

configurations()
そのデバイスの構成群をタプルで返します。

ctrl_transfer(*bmRequestType*, *bRequest*, *wValue=0*, *wIndex=0*, *data_or_wLength=None*, *timeout=None*)
エンドポイント 0 で制御転送 (control transfer) を行います。

このメソッドは、エンドポイント 0 を介して制御転送を発行するために使用します（エンドポイント 0 は常に制御エンドポイントである必要があります)。

パラメータ bmRequestType、bRequest、wValue、wIndex は、USB 標準制御要求 (Standard Control Request) フォーマットと同じです。

Control requests may or may not have a data payload to write/read. In cases which it has, the direction bit of the bmRequestType field is used to infer the desired request direction. For host to device requests (OUT), data_or_wLength parameter is the data payload to send, and it must be a sequence type convertible to an array object. In this case, the return value is the number of bytes written in the data payload. For device to host requests (IN), data_or_wLength is either the wLength parameter of the control request specifying the number of bytes to read in data payload, and the return value is an array object with data read, or an array object which the data will be read to, and the return value is the number of bytes read.

default_timeout
Default timeout for transfer I/O functions

detach_kernel_driver(*interface*)
Detach a kernel driver.

If successful, you will then be able to perform I/O.

The interface parameter is the device interface number to detach the driver from.

get_active_configuration()
Return a Configuration object representing the current configuration set.

is_kernel_driver_active(*interface*)
Determine if there is kernel driver associated with the interface.

If a kernel driver is active, the object will be unable to perform I/O.

The interface parameter is the device interface number to check.

langids

Return the USB device's supported language ID codes.

These are 16-bit codes familiar to Windows developers, where for example instead of en-US you say 0x0409. USB_LANGIDS.pdf on the usb.org developer site for more info. String requests using a LANGID not in this array should not be sent to the device.

This property will cause some USB traffic the first time it is accessed and cache the resulting value for future use.

manufacturer

Return the USB device's manufacturer string descriptor.

This property will cause some USB traffic the first time it is accessed and cache the resulting value for future use.

parent

Return the parent device.

product

Return the USB device's product string descriptor.

This property will cause some USB traffic the first time it is accessed and cache the resulting value for future use.

read(*endpoint*, *size_or_buffer*, *timeout=None*)

Read data from the endpoint.

This method is used to receive data from the device. The endpoint parameter corresponds to the bEndpointAddress member whose endpoint you want to communicate with. The size_or_buffer parameter either tells how many bytes you want to read or supplies the buffer to receive the data (it *must* be an object of the type array).

The timeout is specified in miliseconds.

If the size_or_buffer parameter is the number of bytes to read, the method returns an array object with the data read. If the size_or_buffer parameter is an array object, it returns the number of bytes actually read.

reset()

Reset the device.

serial_number

Return the USB device's serial number string descriptor.

This property will cause some USB traffic the first time it is accessed and cache the resulting value for future use.

set_configuration(*configuration=None*)

Set the active configuration.

The configuration parameter is the bConfigurationValue field of the configuration you want to set as active. If you call this method without parameter, it will use the first configuration found. As a device hardly ever has more than one configuration, calling the method without arguments is enough to get the device ready.

set_interface_altsetting(*interface=None*, *alternate_setting=None*)

Set the alternate setting for an interface.

When you want to use an interface and it has more than one alternate setting, you should call this method to select the appropriate alternate setting. If you call the method without one or the two parameters, it will be selected the first one found in the Device in the same way of the set_configuration method.

Commonly, an interface has only one alternate setting and this call is not necessary. For most devices, either it has more than one alternate setting or not, it is not harmful to make a call to this method with no arguments, as devices will silently ignore the request when there is only one alternate setting, though the USB Spec allows devices with no additional alternate setting return an error to the Host in response to a SET_INTERFACE request.

If you are in doubt, you may want to call it with no arguments wrapped by a try/except clause:

```
>>> try:
>>>     dev.set_interface_altsetting()
>>> except usb.core.USBError:
>>>     pass
```

write(*endpoint*, *data*, *timeout=None*)

Write data to the endpoint.

This method is used to send data to the device. The endpoint parameter corresponds to the bEndpointAddress member whose endpoint you want to communicate with.

The data parameter should be a sequence like type convertible to the array type (see array module).

The timeout is specified in miliseconds.

The method returns the number of bytes written.

class usb.core.**Configuration**(*device*, *configuration=0*)

ベースクラス: object

Represent a configuration object.

This class contains all fields of the Configuration Descriptor according to the USB Specification. You may access them as class properties. For example, to access the field bConfigurationValue of the configuration descriptor, you can do so:

```
>>> import usb.core
>>> dev = usb.core.find()
>>> for cfg in dev:
>>>     print cfg.bConfigurationValue
```

interfaces()

Return a tuple of the configuration interfaces.

set()

Set this configuration as the active one.

class usb.core.**Interface**(*device*, *interface=0*, *alternate_setting=0*, *configuration=0*)

ベースクラス: object

Represent an interface object.

This class contains all fields of the Interface Descriptor according to the USB Specification. You may access them as class properties. For example, to access the field bInterfaceNumber of the interface descriptor, you can do so:

```
>>> import usb.core
>>> dev = usb.core.find()
>>> for cfg in dev:
>>>     for i in cfg:
>>>         print i.bInterfaceNumber
```

endpoints()

Return a tuple of the interface endpoints.

set_altsetting()

Set the interface alternate setting.

class usb.core.**Endpoint**(*device*, *endpoint*, *interface=0*, *alternate_setting=0*, *configuration=0*)

ベースクラス: object

Represent an endpoint object.

This class contains all fields of the Endpoint Descriptor according to the USB Specification. You can access them as class properties. For example, to access the field bEndpointAddress of the endpoint descriptor, you can do so:

```
>>> import usb.core
>>> dev = usb.core.find()
>>> for cfg in dev:
>>>     for i in cfg:
>>>         for e in i:
>>>             print e.bEndpointAddress
```

clear_halt()

Clear the halt/status condition of the endpoint.

read(*size_or_buffer*, *timeout=None*)

Read data from the endpoint.

The parameter size_or_buffer is either the number of bytes to read or an array object where the data will be put in and timeout is the time limit of the operation. The transfer type and endpoint address are automatically inferred.

The method returns either an array object or the number of bytes actually read.

For details, see the Device.read() method.

write(*data*, *timeout=None*)

Write data to the endpoint.

The parameter data contains the data to be sent to the endpoint and timeout is the time limit of the operation. The transfer type and endpoint address are automatically inferred.

The method returns the number of bytes written.

For details, see the Device.write() method.

exception usb.core.**USBError**(*strerror*, *error_code=None*, *errno=None*)

ベースクラス: OSError

Exception class for USB errors.

Backends must raise this exception when USB related errors occur. The backend specific error code is available through the 'backend_error_code' member variable.

exception usb.core.**USBTimeoutError**(*strerror*, *error_code=None*, *errno=None*)

ベースクラス: *usb.core.USBError*

Exception class for connection timeout errors.

Backends must raise this exception when a call on a USB connection returns a timeout error code.

exception usb.core.**NoBackendError**

ベースクラス: ValueError

Exception class when a valid backend is not found.

usb.core.**find**(*find_all=False*, *backend=None*, *custom_match=None*, ***args*)

Find an USB device and return it.

find() is the function used to discover USB devices. You can pass as arguments any combination of the USB Device Descriptor fields to match a device. For example:

find(idVendor=0x3f4, idProduct=0x2009)

will return the Device object for the device with idVendor field equals to 0x3f4 and idProduct equals to 0x2009.

If there is more than one device which matchs the criteria, the first one found will be returned. If a matching device cannot be found the function returns None. If you want to get all devices, you can set the parameter find_all to True, then find will return an iterator with all matched devices. If no matching device is found, it will return an empty iterator. Example:

for printer in find(find_all=True, bDeviceClass=7): print (printer)

This call will get all the USB printers connected to the system. (actually may be not, because some devices put their class information in the Interface Descriptor).

You can also use a customized match criteria:

dev = find(custom_match = lambda d: d.idProduct=0x3f4 and d.idvendor=0x2009)

A more accurate printer finder using a customized match would be like so:

def is_printer(dev): import usb.util if dev.bDeviceClass == 7:

return True

for cfg in dev:

if usb.util.find_descriptor(cfg, bInterfaceClass=7) is not None: return True

for printer in find(find_all=True, custom_match = is_printer): print (printer)

Now even if the device class code is in the interface descriptor the printer will be found.

You can combine a customized match with device descriptor fields. In this case, the fields must match and the custom_match must return True. In the our previous example, if we would like to get all printers belonging to the manufacturer 0x3f4, the code would be like so:

printers = list(find(find_all=True, idVendor=0x3f4, custom_match=is_printer))

If you want to use find as a 'list all devices' function, just call it with find_all = True:

devices = list(find(find_all=True))

Finally, you can pass a custom backend to the find function:

find(backend = MyBackend())

PyUSB has builtin backends for libusb 0.1, libusb 1.0 and OpenUSB. If you do not supply a backend explicitly, find() function will select one of the predefineds backends according to system availability.

Backends are explained in the usb.backend module.

usb.core.**show_devices**(*verbose=False*, ***kwargs*)

Show information about connected devices.

The verbose flag sets to verbose or not. **kwargs are passed directly to the find() function.

5.5 usb.legacy モジュール

class usb.legacy.**Bus**(*devices*)

ベースクラス: object

Bus object.

class usb.legacy.**Configuration**(*cfg*)

ベースクラス: object

Configuration descriptor object.

class usb.legacy.**Device**(*dev*)

ベースクラス: object

Device descriptor object

open()

Open the device for use.

Returns a DeviceHandle object

class usb.legacy.**DeviceHandle**(*dev*)

ベースクラス: usb._objfinalizer.AutoFinalizedObject

bulkRead(*endpoint*, *size*, *timeout=100*)

Performs a bulk read request to the endpoint specified.

パラメータ

- **endpoint** – endpoint number.
- **size** – number of bytes to read.
- **timeout** – operation timeout in milliseconds. (default: 100)

Returns a tuple with the data read.

bulkWrite(*endpoint*, *buffer*, *timeout=100*)

Perform a bulk write request to the endpoint specified.

パラメータ

- **endpoint** – endpoint number.

- **buffer** – sequence data buffer to write. This parameter can be any sequence type.
- **timeout** – operation timeout in milliseconds. (default: 100)

Returns the number of bytes written.

claimInterface(*interface*)

Claims the interface with the Operating System.

パラメータ **interface** – interface number or an Interface object.

clearHalt(*endpoint*)

Clears any halt status on the specified endpoint.

パラメータ **endpoint** – endpoint number.

controlMsg(*requestType*, *request*, *buffer*, *value=0*, *index=0*, *timeout=100*)

Perform a control request to the default control pipe on a device.

パラメータ

- **requestType** – specifies the direction of data flow, the type of request, and the recipient.
- **request** – specifies the request.
- **buffer** – if the transfer is a write transfer, buffer is a sequence with the transfer data, otherwise, buffer is the number of bytes to read.
- **value** – specific information to pass to the device. (default: 0) index: specific information to pass to the device. (default: 0)
- **timeout** – operation timeout in milliseconds. (default: 100)

Returns the number of bytes written.

detachKernelDriver(*interface*)

Detach a kernel driver from the interface (if one is attached, we have permission and the operation is supported by the OS)

パラメータ **interface** – interface number or an Interface object.

getDescriptor(*desc_type*, *desc_index*, *length*, *endpoint=-1*)

Retrieves a descriptor from the device identified by the type and index of the descriptor.

パラメータ

- **desc_type** – descriptor type.
- **desc_index** – index of the descriptor.

- **len** – descriptor length.
- **endpoint** – ignored.

getString(*index*, *length*, *langid=None*)

Retrieve the string descriptor specified by index and langid from a device.

パラメータ

- **index** – index of descriptor in the device.
- **length** – number of bytes of the string (ignored)
- **langid** – Language ID. If it is omitted, the first language will be used.

interruptRead(*endpoint*, *size*, *timeout=100*)

Performs a interrupt read request to the endpoint specified.

パラメータ

- **endpoint** – endpoint number.
- **size** – number of bytes to read.
- **timeout** – operation timeout in milliseconds. (default: 100)

Returns a tuple with the data read.

interruptWrite(*endpoint*, *buffer*, *timeout=100*)

Perform a interrupt write request to the endpoint specified.

パラメータ

- **endpoint** – endpoint number.
- **buffer** – sequence data buffer to write. This parameter can be any sequence type.
- **timeout** – operation timeout in milliseconds. (default: 100)

Returns the number of bytes written.

releaseInterface()

Release an interface previously claimed with claimInterface.

reset()

Reset the specified device by sending a RESET down the port it is connected to.

resetEndpoint(*endpoint*)

Reset all states for the specified endpoint.

パラメータ **endpoint** – endpoint number.

setAltInterface(*alternate*)

Sets the active alternate setting of the current interface.

パラメータ **alternate** – an alternate setting number or an Interface object.

setConfiguration(*configuration*)

Set the active configuration of a device.

パラメータ **configuration** – a configuration value or a Configuration object.

class usb.legacy.**Endpoint**(*ep*)

ベースクラス: object

Endpoint descriptor object.

class usb.legacy.**Interface**(*intf*)

ベースクラス: object

Interface descriptor object.

usb.legacy.**busses**()

Returns a tuple with the usb busses.

5.6 usb.libloader モジュール

exception usb.libloader.**LibraryException**

ベースクラス: OSError

exception usb.libloader.**LibraryNotFoundException**

ベースクラス: *usb.libloader.LibraryException*

exception usb.libloader.**NoLibraryCandidatesException**

ベースクラス: *usb.libloader.LibraryNotFoundException*

exception usb.libloader.**LibraryNotLoadedException**

ベースクラス: *usb.libloader.LibraryException*

exception usb.libloader.**LibraryMissingSymbolsException**

ベースクラス: *usb.libloader.LibraryException*

usb.libloader.**locate_library**(*candidates*, *find_library=<function find_library>*)

Tries to locate a library listed in candidates using the given find_library() function (or ctypes.util.find_library). Returns the first library found, which can be the library's name or the path to the library file, depending on find_library(). Returns None if no library is found.

arguments: * candidates – iterable with library names * find_library – function that takes one positional arg (candidate)

and returns a non-empty str if a library has been found. Any "false" value (None,False,empty str) is interpreted as "library not found". Defaults to ctypes.util.find_library if not given or None.

usb.libloader.**load_library**(*lib*, *name=None*, *lib_cls=None*)

Loads a library. Catches and logs exceptions.

Returns: the loaded library or None

arguments: * lib – path to/name of the library to be loaded * name – the library's identifier (for logging)

Defaults to None.

- lib_cls – library class. Defaults to None (-> ctypes.CDLL).

usb.libloader.**load_locate_library**(*candidates*, *cygwin_lib*, *name*, *win_cls=None*, *cygwin_cls=None*, *others_cls=None*, *find_library=None*, *check_symbols=None*)

Locates and loads a library.

Returns: the loaded library

arguments: * candidates – candidates list for locate_library() * cygwin_lib – name of the cygwin library * name – lib identifier (for logging). Defaults to None. * win_cls – class that is used to instantiate the library on

win32 platforms. Defaults to None (-> ctypes.CDLL).

- cygwin_cls – library class for cygwin platforms. Defaults to None (-> ctypes.CDLL).
- others_cls – library class for all other platforms. Defaults to None (-> ctypes.CDLL).
- find_library – see locate_library(). Defaults to None.
- check_symbols – either None or a list of symbols that the loaded lib must provide (hasattr(<>)) in order to be considered valid. LibraryMissingSymbolsException is raised if any symbol is missing.

raises: * NoLibraryCandidatesException * LibraryNotFoundException * LibraryNotLoadedException * LibraryMissingSymbolsException

5.7 usb.util モジュール

usb.util - ユーティリティ関数群。

本モジュールからは以下がエスクポートされます:

endpoint_address - return the endpoint absolute address. endpoint_direction - return the endpoint transfer direction. endpoint_type - return the endpoint type ctrl_direction - return the direction of a control transfer build_request_type - build a bmRequestType field of a control transfer. find_descriptor - find an inner descriptor. claim_interface - explicitly claim an interface. release_interface - explicitly release an interface. dispose_resources - release internal resources allocated by the object. get_langids - retrieve the list of supported string languages from the device. get_string - retrieve a string descriptor from the device.

usb.util.**build_request_type**(*direction*, *type*, *recipient*)

Build a bmRequestType field for control requests.

These is a conventional function to build a bmRequestType for a control request.

The direction parameter can be CTRL_OUT or CTRL_IN. The type parameter can be CTRL_TYPE_STANDARD, CTRL_TYPE_CLASS, CTRL_TYPE_VENDOR or CTRL_TYPE_RESERVED values. The recipient can be CTRL_RECIPIENT_DEVICE, CTRL_RECIPIENT_INTERFACE, CTRL_RECIPIENT_ENDPOINT or CTRL_RECIPIENT_OTHER.

Return the bmRequestType value.

usb.util.**claim_interface**(*device*, *interface*)

Explicitly claim an interface.

PyUSB users normally do not have to worry about interface claiming, as the library takes care of it automatically. But there are situations where you need deterministic interface claiming. For these uncommon cases, you can use claim_interface.

If the interface is already claimed, either through a previously call to claim_interface or internally by the device object, nothing happens.

usb.util.**create_buffer**(*length*)

Create a buffer to be passed to a read function.

A read function may receive an out buffer so the data is read inplace and the object can be reused, avoiding the overhead of creating a new object at each new read call. This function creates a compatible sequence buffer of the given length.

usb.util.**ctrl_direction**(*bmRequestType*)

Return the direction of a control request.

The bmRequestType parameter is the value of the bmRequestType field of a control transfer. The possible return values are CTRL_OUT or CTRL_IN.

usb.util.**dispose_resources**(*device*)

Release internal resources allocated by the object.

Sometimes you need to provide deterministic resources freeing, for example to allow another application to talk to the device. As Python does not provide deterministic destruction, this function releases all internal resources

allocated by the device, like device handle and interface policy.

After calling this function, you can continue using the device object normally. If the resources will be necessary again, it will be allocated automatically.

usb.util.**endpoint_address**(*address*)

Return the endpoint absolute address.

The address parameter is the bEndpointAddress field of the endpoint descriptor.

usb.util.**endpoint_direction**(*address*)

Return the endpoint direction.

The address parameter is the bEndpointAddress field of the endpoint descriptor. The possible return values are ENDPOINT_OUT or ENDPOINT_IN.

usb.util.**endpoint_type**(*bmAttributes*)

Return the transfer type of the endpoint.

The bmAttributes parameter is the bmAttributes field of the endpoint descriptor. The possible return values are: ENDPOINT_TYPE_CTRL, ENDPOINT_TYPE_ISO, ENDPOINT_TYPE_BULK or ENDPOINT_TYPE_INTR.

usb.util.**find_descriptor**(*desc*, *find_all=False*, *custom_match=None*, ***args*)

Find an inner descriptor.

find_descriptor works in the same way as the core.find() function does, but it acts on general descriptor objects. For example, suppose you have a Device object called dev and want a Configuration of this object with its bConfigurationValue equals to 1, the code would be like so:

```
>>> cfg = util.find_descriptor(dev, bConfigurationValue=1)
```

You can use any field of the Descriptor as a match criteria, and you can supply a customized match just like core.find() does. The find_descriptor function also accepts the find_all parameter to get an iterator instead of just one descriptor.

usb.util.**get_langids**(*dev*)

Retrieve the list of supported Language IDs from the device.

Most client code should not call this function directly, but instead use the langids property on the Device object, which will call this function as needed and cache the result.

USB LANGIDs are 16-bit integers familiar to Windows developers, where for example instead of en-US you say 0x0409. See the file USB_LANGIDS.pdf somewhere on the usb.org site for a list, which does not claim to be complete. It requires "system software must allow the enumeration and selection of LANGIDs that are not currently on this list." It also requires "system software should never request a LANGID not defined in the

LANGID code array (string index = 0) presented by a device." Client code can check this tuple before issuing string requests for a specific language ID.

dev is the Device object whose supported language IDs will be retrieved.

The return value is a tuple of integer LANGIDs, possibly empty if the device does not support strings at all (which USB 3.1 r1.0 section 9.6.9 allows). In that case client code should not request strings at all.

A USBError may be raised from this function for some devices that have no string support, instead of returning an empty tuple. The accessor for the langids property on Device catches that case and supplies an empty tuple, so client code can ignore this detail by using the langids property instead of directly calling this function.

usb.util.**get_string**(*dev*, *index*, *langid=None*)

Retrieve a string descriptor from the device.

dev is the Device object which the string will be read from.

index is the string descriptor index and langid is the Language ID of the descriptor. If langid is omitted, the string descriptor of the first Language ID will be returned.

Zero is never the index of a real string. The USB spec allows a device to use zero in a string index field to indicate that no string is provided. So the caller does not have to treat that case specially, this function returns None if passed an index of zero, and generates no traffic to the device.

The return value is the unicode string present in the descriptor, or None if the requested index was zero.

usb.util.**release_interface**(*device*, *interface*)

Explicitly release an interface.

This function is used to release an interface previously claimed, either through a call to claim_interface or internally by the device object.

Normally, you do not need to worry about claiming policies, as the device object takes care of it automatically.

5.8 モジュール内容

PyUSB - Python から簡単に USB にアクセスできるようにします。

本パッケージはイカのモジュールとサブパッケージをエクスポートします:

core - 主たる USB 実装、 legacy - 0.x バージョン利用者互換レイヤ、backend - バックエンド実装のサポート、 control - USB 標準制御要求 (standard control requests)、 libloader - バックエンドライブラリ読み込みのためのヘルパーモジュール

バージョン 1.0 以降では、メインの PyUSB 実装は「usb.core」モジュールにあります。 新しいアプリケーションではそれを使用することをお勧めします。

第 6 章

索引

- genindex
- modindex
- search

Python モジュール索引

u

索引