SimplicityGuy Updating URLs to reflect move to pyusb organization.

174b9c3 on Nov 12, 2017

8 contributors 

  

641 lines (500 sloc) 26.2 KB

Programming with PyUSB 1.0

Let me introduce myself

PyUSB 1.0 is a Python library allowing easy USB access. PyUSB provides several features:

100% written in Python:

Unlike the 0.x version, which is written in C, 1.0 version is written in Python. This allows Python programmers with no background in C to understand better how PyUSB works.

Platform neutrality:

1.0 version implements a frontend-backend scheme. This isolates the API from system specific implementation details. The glue between the two layers is the `IBackend` interface. PyUSB comes with builtin backends for libusb 0.1, libusb 1.0 and OpenUSB. You can write your own backend if you desire to.

Portability:
PyUSB should run on any platform with Python >= 2.4, ctypes and at least one of the supported builtin backends.

Easiness:
Communicating with an USB device has never been so easy! USB is a complex protocol, but PyUSB has good defaults for most common configurations.

Support for isochronous transfers:
PyUSB supports isochronous transfers if the underlying backend supports it.

Although PyUSB makes USB programming less painful, it is assumed in this tutorial that you have a minimal USB protocol background. If you don't know anything about USB, I recommend you the excellent Jan Axelson's book **USB Complete**.

Enough talk, let's code!

Who's who

First of all, let's give an overview on the PyUSB modules. PyUSB modules are under the `usb` package, with the following modules:

Content	Description
core	The main USB module.
util	Utility functions.
control	Standard control requests.
legacy	The 0.x compatibility layer.
backend	A subpackage containing the builtin backends.

For example, to import the `core` module, type the following:

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

Let's get it started

Following is a simplistic program that sends the 'test' string to the first OUT endpoint found:

```
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,
    # 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')
```

The first two lines import PyUSB package modules. `usb.core` is the main module, and `usb.util` contains utility functions. The next command searches for our device and returns an instance object if it is found. If not, `None` is returned. After, we set the configuration to use. Note that no argument indicating what configuration we want was supplied. As you will see, many PyUSB functions have defaults for most common devices. In this case, the configuration set is the first one found.

Then, we look for the endpoint we are interested. We search for it inside the first interface we have. After finding the endpoint, we send the data to it.

If we know the endpoint address in advance, we could just call the `write` function from the device object:

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

Here we write the string 'test' at the endpoint address *1*. All these functions will be detailed in the following sections.

What's wrong?

Every function in PyUSB raises an exception in case of an error. Besides the Python standard exceptions, PyUSB defines the `usb.core.USBError` for USB related errors.

You can also use the PyUSB log functionality. It uses the logging module. To enable it, define the environment variable `PYUSB_DEBUG` with one of the following level names: `critical`, `error`, `warning`, `info` or `debug`.

By default the messages are sent to sys.stderr. If you want to, you can redirect log messages to a file by defining the `PYUSB_LOG_FILENAME` environment variable. If its value is a valid file path, messages will be written to it, otherwise it will be sent to `sys.stderr`.

Where are you?

The `find()` function in the `core` module is used to find and enumerate devices connected to the system. For example, let's say that our device has a vendor ID equal to 0xfffe and product ID equals to 0x0001. If we would like to find it, we proceed in this way:

```
import usb.core

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

That's it, the function will return an `usb.core.Device` object representing our device. If the device is not found, it returns `None`. Actually, you can use any field of the Device Descriptor you desire. For example, what if we would like to discover if there is a USB printer connected to the system? This is very easy:

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

The 7 is the code for the printer class according to the USB spec. Hey, wait, what if I want to enumerate all printers present? No problem:

```
# 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.')
```

What happened? Well, it is time for a little explanation... `find` has a parameter called `find_all` that defaults to False. When it is false [1], `find` will return the first device found that matches the specified criteria (more on that soon). If you give it a *true* value, `find` will instead return a list with all devices matching the criteria. That's it! Simple, isn't it?

Finished? No! I have not told you the whole history: many devices actually put their class information in the Interface Descriptor instead of the Device Descriptor. So, to really find all printers connected to the system, we would need to transverse all configurations, and then all interfaces and check if one of the interfaces has its bInterfaceClass field equal to 7. If you are a programmer like me, you might be wondering if there is an easier way to do that. The answer is yes, there is. First, let's give a look on the final code to find all printers connected:

```
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))
```

The `custom_match` parameter accepts any callable object that receives the device object. It must return true for a matching device, and false for a non-matching device. You can also combine `custom_match` with device fields if you want:

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

Here we are only interested in the printers of the 0xfffe vendor.

Describe yourself

Ok, we've found our device, but before talking to it, we would like to know more about it, you know, configurations, interfaces, endpoints, transfer types...

If you have a device, you can access any device descriptor fields as object properties:

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

To access the configurations available in the device, you can iterate over the device:

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

In the same way, you can iterate over a configuration to access the interfaces, and iterate over the interfaces to access their endpoints. Each kind of object has as attributes the fields of the respective descriptor. Let's see an example:

```
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')
```

You can also use the subscript operator to access the descriptors randomly, like this:

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

As you can see, the index is zero-based. But wait! There is something weird in the way I access an interface... Yes, you are right, the subscript operator in the Configuration accepts a sequence of two items, with the first one being the index of the Interface and the second one, the alternate setting. So, to access the first interface, but its second alternate setting, we write `cfg[(0,1)]`.

Now it's time to we learn a powerful way to find descriptors, the `find_descriptor` utility function. We have already seen it in the printer finding example. `find_descriptor` works in almost the same way as `find`, with two exceptions:

- `find_descriptor` receives as its first parameter the parent descriptor that you will search on.
- There is no `backend` [2] parameter.

For example, if we have a configuration descriptor `cfg` and want to find all alternate settings of the interface 1, we do so:

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

Notice that `find_descriptor` is in the `usb.util` module. It also accepts the early described `custom_match` parameter.

Dealing with multiple identical devices

Sometimes you may have two identical devices connected to the computer. How can you differentiate them? `Device` objects come with two additional attributes which are not part of the USB Spec, but are very useful: `bus` and `address` attributes. First of all, it is worth it to say that these attributes come from the backend and a backend is free to not support them, in which case they are set to `None`. That said, these attributes represent the bus number and bus address of the device and, as you might already have imagined, can be used to differentiate two devices with the same `idVendor` and `idProduct` attributes.

How am I supposed to work?

USB devices after connection must be configured through a few standard requests. When I started to study USB spec, I found myself confused with descriptors, configurations, interfaces, alternate settings, transfer types and all this stuff... And worst, you cannot simply ignore them, a device does not work without setting a configuration, even if it has just one! PyUSB tries to make your life as easy as possible. For example, after getting your device object, one of the first things you need to do before communicating with it is issuing a `set_configuration` request. The parameter for this request is the `bConfigurationValue` of the configuration you are interested on. Most devices have no more than one configuration, and tracking the configuration value to use is annoying (although most code I have seen simply hardcodes it). Therefore, in PyUSB, you can just issue a `set_configuration` call with no arguments. In this case, it will set the first configuration found (if your device has just one, you don't need to worry about the configuration value at all). For example, let's imagine you have a device with one configuration descriptor with its bConfigurationValue field equals to 5 [3], the following calls below will work equally:

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

Wow! You can use a `Configuration` object as a parameter to `set_configuration`! Yes, and also it has a `set` method to configure itself as the current configuration.

The other setting you might or might not have to configure is the interface alternate setting. Each device can have only one activated configuration at a time, and each configuration may have more than one interface, and you can use all interfaces at the same time. You better understand this concept if you think of an interface as a logical device. For example, let's imagine a multifunction printer, which is at the same time a printer and a scanner. To keep things simple (or at least as simple as we can), let's consider that it has just one configuration. As we have a printer and a scanner, the configuration has two interfaces, one for the printer and one for the scanner. A device with more than one interface is called a composite device. When you connect your multifunction printer to your computer, the Operating System would load two different drivers: one for each "logical" peripheral you have [4].

What about the alternate setting? Good you asked. An interface has one or more alternate settings. An interface with just one alternate setting is considered to not having an alternate setting [5]. Alternate settings are for interfaces what configurations are for devices, i.e, for each interface, you can have only one alternate setting active. For example, USB spec says that a device cannot have an isochronous endpoint in its primary alternate setting [6], so a streaming device must have at least two alternate settings, with the second one having the isochronous endpoint(s). But as opposed to configurations, interfaces with just one alternate setting don't need to be set [7]. You select an interface alternate setting through the `set_interface_altsetting` function:

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

Warning

The USB spec says that a device is allowed to return an error in case it receives a SET_INTERFACE request for an interface that has no additional alternate settings. So, if you are not sure if either the interface has more than one alternate setting or it accepts a SET_INTERFACE request, the safest way is to call `set_interface_altsetting` inside an try-except block, like this:

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

You can also use an `Interface` object as parameter to the function, the `interface` and `alternate_setting` parameters are automatically inferred from `bInterfaceNumber` and `bAlternateSetting` fields. Example:

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

Warning

The `Interface` object must belong to the active configuration descriptor.

Talk to me, honey

Now it's time for us to learn how to communicate with USB devices. USB has four flavors of transfers: bulk, interrupt, isochronous and control. I don't intend to explain the purpose of each transfer and the differences among them. Therefore, I assume you know at least the basics of the USB transfers.

Control transfer is the only transfer that has structured data described in the spec, the others just send and receive raw data from USB point of view. Because of it, you have a different function to deal with control transfers, all the other transfers are managed by the same functions.

You issue a control transfer through the `ctrl_transfer` method. It is used both for OUT and IN transfers. The transfer direction is determined from the `bmRequestType` parameter.

The `ctrl_transfer` parameters are almost equal to the control request structure. Following is a example of how to do a control transfer [8]:

```
>>> msg = 'test'
>>> assert dev.ctrl_transfer(0x40, CTRL_LOOPBACK_WRITE, 0, 0, msg) == le
>>> ret = dev.ctrl_transfer(0xC0, CTRL_LOOPBACK_READ, 0, 0, len(msg))
>>> sret = ''.join([chr(x) for x in ret])
>>> assert sret == msg
```

In this example, it is assumed that our device implements two custom control requests that act as a loopback pipe. What you write with the `CTRL_LOOPBACK_WRITE` message, you can read with the `CTRL_LOOPBACK_READ` message.

The first four parameters are the `bmRequestType`, `bmRequest`, `wValue` and `wIndex` fields of the standard control transfer structure. The fifth parameter is either the data payload for an OUT transfer or the number of bytes to read in an IN transfer. The data payload can be any sequence type that can be used as a parameter for the array `__init__` method. If there is no data payload, the parameter should be `None` (or 0 in case of an IN transfer). There is one last optional parameter specifying the timeout of the operation. If you don't supply it, a default timeout will be used (more on that later). In an OUT transfer, the return value is the number of bytes really sent to the device. In an IN transfer, the return value is an array object with the data read.

For the other transfers, you use the methods `write` and `read`, respectively, to write and read data. You don't need to worry about the transfer type, it is automatically determined from the endpoint address. Here is our loopback example assuming the we have a loopback pipe in the endpoint 1:

```
>>> 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
```

The first and third parameters are equal for both methods, they are the endpoint address and timeout, respectively. The second parameter is the data payload (write) or the number of bytes to read (read). The returned data if either an instance of the array object for the `read` method or the number of bytes written for the `write` method.

Since beta 2 version, instead of the number of bytes, you can also pass to `read` and `ctrl_transfer` an array object in which the data will be read into. In this case, the number of bytes to read will be the length of the array times the `array.itemsize` value.

As in `ctrl_transfer`, the `timeout` parameter is optional. When the `timeout` is omitted, it is used the `Device.default_timeout` property as the operation timeout.

Control yourself

Besides the transfers functions, the module `usb.control` offers functions which implement the standard USB control requests and the `usb.util` module has the convenience function `get_string` specifically to return string descriptors.

Additional Topics

Behind every great abstraction, there's a great implementation

In the early days, there was only libusb. Then came libusb 1.0, and we had libusb 0.1 and 1.0. After, they created OpenUSB, and now we live at the Tower of Babel of the USB libraries [9]. How does PyUSB deal with it? Well, PyUSB is a democratic library, you may choose whichever library you want. Actually, you can write your own USB library from scratch and tell PyUSB to use it.

The `find` function has one more parameter that I haven't told you. It is the `backend` parameter. If you don't supply it, it will be used one of the builtin backends. A backend is an object inherited from `usb.backend.IBackend`, responsible to implement the operating system specific USB stuff. As you might guess, the builtins are libusb 0.1, libusb 1.0 and OpenUSB backends.

You can create your own backend and use it. Just inherit from `IBackend` and implement the methods necessary. You might want to take a look at the `usb.backend` package documentation to learn how to do that.

Don't be selfish

Python has what we call *automatic memory management*. This means that the virtual machine will decide when to release objects from the memory. Under the hood, PyUSB manages all low level resources it needs to work (interface claiming, device handles, etc.) and most of the users don't need to worry about that. But, because of the nondeterministic nature of automatic object destruction of Python, users cannot predict when the resources allocated will be released. Some applications need to allocate and free the resources deterministically. For these kind of applications, the `usb.util` module has a set of functions to deal with resource management.

If you want to claim and release interfaces manually, you may use the `claim_interface` and `release_interface` functions. `claim_interface` will claim the specified interface if the device has not done it yet. If the device already claimed the interface, it does nothing. In a similar way, `release_interface` will release the specified interface if it is claimed. If the interface is not claimed, it does nothing. You can use manual interface claim to solve the configuration selection problem described in the libusb documentation.

If you want to free all resources allocated by the device object (including interfaces claimed), you can use the `dispose_resources` function. It releases all resources allocated and puts the device object (but not the device hardware itself) in the state it was at the time when the `find` function returned.

Specifying libraries by hand

In general, a backend is an wrapper on a shared library which implements the USB access API. By default, the backend uses the find_library() ctypes function. On Linux and other Unix like Operating Systems, `find_library` tries to run external programs (like */sbin/ldconfig*, *gcc* and *objdump*) to find the library file.

On systems where these programs are missing and/or the library cache is disabled, this function cannot be used. To overcome this limitation, PyUSB allows you to supply a custom find_library() function to the backend.

An example for such scenario would be:

```
>>> import usb.core
>>> import usb.backend.libusb1
>>>
>>> backend = usb.backend.libusb1.get_backend(find_library=lambda x: "/u
>>> dev     = usb.core.find(..., backend=backend)
```

Notice the find_library argument for the get_backend() function, in which you supply a function that is responsible to find the correct library for the backend.

Old school rules

If you wrote an application using the old PyUSB API (0.whatever), you may be asking yourself if you need to update your code to use the new API. Well, you should, but you don't need to. PyUSB 1.0 comes with the `usb.legacy` compatibility module. It implements the older API above the new API. "So, do I have just to replace my `import usb` statement with `import usb.legacy as usb` to get my application working?", you ask. The answer is yes, it will work, but you don't have to. If you run your application untouched it will just work, because the `import usb` statement will import all public symbols from `usb.legacy` . If you face a problem, probably you found a bug.

Help me, please

If you need help, **do not email me**, the mailing list is there for this. Subscribe instructions can be found at the PyUSB website.

[1]	When I say True or False (capitalized), I mean the respective values of the Python language. And when I say true and false, I mean any expression in Python which evals to true or false.

[2]	See backend specific documentation.
[3]	USB spec does not impose any sequential value to the configuration value. The same is true for interface and alternate setting numbers.
[4]	Actually things are a little more complex, but this simple explanation is enough for us.
[5]	I know it sounds weird.
[6]	This is because if there is no bandwidth for isochronous transfer at the device configuration time, the device can be successfully enumerated.
[7]	This does not happen for configurations because a device is allowed to be in an unconfigured state.
[8]	In PyUSB, control transfers are only issued in the endpoint 0. It's very very very rare a device having an alternate control endpoint (I've never seen such a device).
[9]	It's just a joke, don't take it seriously. Many choices is better than no choice.