pylibftdi Documentation

Release 0.21.0

Ben Bass

Jun 05, 2023

Contents

pylibftdi is a simple library interacting with FTDI devices to provide serial and parallel IO from Python.

Examples:

```
>>> from pylibftdi import BitBangDevice
>>> with BitBangDevice('FT0123') as dev:
...     dev.port |= 1
```

```
>>> # Send a MIDI 'note on' message
>>> from pylibftdi import Device
>>> with Device() as dev:
...     dev.baudrate = 31250
...     dev.write('\x90\x64\x64')
```

The two main use cases it serves are:

- the need to control or monitor external equipment, for which a FTDI module may be a cheap and reliable starting point.
- the need to interact with existing devices which are known to contain FTDI chipsets for their USB interface.

FTDI (http://www.ftdichip.com) create devices (chipsets, modules, cables etc) to interface devices to the USB port of your computer.

libftdi (http://www.intra2net.com/en/developer/libftdi/) is an open source driver to communicate with these devices, and runs on top of libusb. It works on Windows, Linux, and Mac OS X, and likely other systems too.

pylibftdi is a pure Python module which interfaces (via ctypes) to libftdi, exposing a simple file-like API to connected devices. It supports serial and parallel IO in a straight-forward way, and aims to be one of the simplest ways of interacting with the world outside your PC.

CHAPTER 1

Contents

1.1 Introduction

pylibftdi is a minimal Pythonic interface to FTDI devices using libftdi[1].

Features

- No dependencies beyond standard library and a *libftdi* install.
- Supports parallel and serial devices
- Support for multiple devices
- File-like interface wherever appropriate
- Cross-platform

Limitations

- The API might change prior to reaching a 1.0 release.

1.1.1 Usage

The primary interface is the `Device` class in the pylibftdi package; this gives serial access on relevant FTDI devices (e.g. the UM232R), providing a file-like interface (read, write). Baudrate is controlled with the `baudrate` property.

If a Device instance is created with `mode='t'` (text mode) then read() and write() can use the given `encoding` (defaulting to latin-1). This allows easier integration with passing unicode strings between devices.

Multiple devices are supported by passing the desired device serial number (as a string) in the `device_id` parameter - this is the first parameter in both Device() and BitBangDevice() constructors. Alternatively the device 'description' can be given, and an attempt will be made to match this if matching by serial number fails.

[1] http://www.intra2net.com/en/developer/libftdi/

Examples

```
>>> from pylibftdi import Device
>>>
>>> with Device(mode='t') as dev:
...     dev.baudrate = 115200
...     dev.write('Hello World')
```

The pylibftdi.BitBangDevice wrapper provides access to the parallel IO mode of operation through the `port` and `direction` properties. These provide an 8 bit IO port including all the relevant bit operations to make things simple.

```
>>> from pylibftdi import BitBangDevice
>>>
>>> with BitBangDevice('FTE00P4L') as bb:
...     bb.direction = 0x0F  # four LSB are output(1), four MSB are␣
↪input(0)
...     bb.port |= 2         # set bit 1
...     bb.port &= 0xFE      # clear bit 0
```

There is support for a number of external devices and protocols, including interfacing with HD44780 LCDs using the 4-bit interface.

1.1.2 History & Motivation

This package is the result of various bits of work using FTDI's devices, primarily for controlling external devices. Some of this is documented on the codedstructure blog, codedstructure.blogspot.com

Several other open-source Python FTDI wrappers exist, and each may be best for some projects. Some aim at closely wrapping the libftdi interface, others use FTDI's own D2XX driver (ftd2xx[2]) or talk directly to USB via libusb or similar (such as pyftdi[3]).

The aim for pylibftdi is to work with libftdi, but to provide a high-level Pythonic interface. Various wrappers and utility functions are also part of the distribution; following Python's batteries included approach, there are various interesting devices supported out-of-the-box - or at least there will be soon!

1.1.3 Plans

- Add more examples: SPI devices, knight-rider effects, input devices, MIDI...
- Perhaps add support for D2XX driver, though the name then becomes a slight liability ;)

1.1.4 License

[2] http://pypi.python.org/pypi/ftd2xx
[3] https://github.com/eblot/pyftdi

1.2 Quick Start

1.2.1 Install pylibftdi

See the installation instructions for more detailed requirements, but hopefully things will work by just running the following:

```
$ python3 -m pip install pylibftdi
```

1.2.2 Connect and enumerate FTDI devices

Connect the FTDI device to a free USB port. Run the `list_devices` example to enumerate connected FTDI devices:

```
$ python3 -m pylibftdi.examples.list_devices
```

For each connected device, this will show manufacturer, model identifier, and serial number. With a single device connected, the output maybe something like the following:

```
FTDI:UM232H:FTUBIOWF
```

Though hopefully with a different serial number, or else you've either stolen mine, or you are me...

1.2.3 Test some actual IO

Output example

Connect an LED between D0 of your bit-bang capable device and ground, via a 330 - 1K ohm resistor as appropriate.

Test the installation and functioning of pylibftdi with the following:

```
$ python3 -m pylibftdi.examples.led_flash
```

The LED should now flash at approximately 1Hz.

Input example

To test some input, remove any connections from the port lines initially, then run the following, which reads and prints the status of the input lines regularly:

```
$ python3 -m pylibftdi.examples.pin_read
```

The `pin_read` example is a complete command line application which can be used to monitor for particular values on the attached device pins, and output an appropriate error code on match. Repeat the above with a trailing `--help` for info.

Using `pylibftdi` from the REPL

Since `pylibftdi` v0.18.0, a `__main__.py` module is included which imports all the exported constants, classes and functions from `pylibftdi`.

This allows quick interaction with FTDI devices from the Python REPL:

```
$ python3 -im pylibftdi
>>> d = Device()
>>> d.write('Hello World')
>>>
```

1.3 Installation

Unsurprisingly, `pylibftdi` depends on `libftdi`, and installing this varies according to your operating system. Chances are that following one of the following instructions will install the required prerequisites. If not, be aware that libftdi in turn relies on `libusb`.

Installing `pylibftdi` itself is straightforward - it is a pure Python package (using `ctypes` for bindings), and has no dependencies outside the Python standard library for installation. Don't expect it to work happily without `libftdi` installed though :-)

```
$ pip install pylibftdi
```

Depending on your environment, you may want to set up a virtual environment[4] or use either the `--user` flag, or prefix the command with `sudo` to gain root privileges.

1.3.1 Windows

I perform only limited testing of pylibftdi on Windows, but it should work correctly provided the requirements of libftdi and libusb are correctly installed.

Recent libftdi binaries for Windows seem to be available from the picusb[5] project on Sourceforge. Download libftdi1-1.1_devkit_x86_x64_21Feb2014.zip or later from that site, which includes the required

Installing libraries on Windows is easier with recent versions of Python (2.7.9, 3.4+) installing *pip* directly, so the standard approach of *pip install pylibftdi* will now easily work on Windows.

1.3.2 Mac OS X

I suggest using homebrew[6] to install libftdi:

```
$ brew install libftdi
```

On OS X Mavericks (and presumably future versions) Apple include a driver for FTDI devices. This needs unloading before `libftdi` can access FTDI devices directly. See the Troubleshooting section for instructions.

[4] https://packaging.python.org/guides/installing-using-pip-and-virtual-environments/
[5] http://sourceforge.net/projects/picusb/files/
[6] http://mxcl.github.com/homebrew/

1.3.3 Linux

There are two steps in getting a sensible installation in Linux systems:

1. Getting `libftdi` and its dependencies installed
2. Ensuring permissions allow access to the device without requiring root privileges. Symptoms of this not being done are programs only working properly when run with `sudo`, giving '-4' or '-8' error codes in other cases.

Each of these steps will be slightly different depending on the distribution in use. I've tested `pylibftdi` on Debian Wheezy (on a Raspberry Pi), Ubuntu (various versions, running on a fairly standard ThinkPad laptop), and Arch Linux (running on a PogoPlug - one of the early pink ones).

Debian (Raspberry Pi) / Ubuntu etc

On Debian like systems (including Ubuntu, Mint, Debian, etc), the package `libftdi1-dev` should give you what you need as far as the libftdi library is concerned:

```
$ sudo apt-get install libftdi1-dev
```

The following works for both a Raspberry Pi (Debian Wheezy) and Ubuntu 12.04, getting ordinary users (e.g. 'pi' on the RPi) access to the FTDI device without needing root permissions:

1. Create a file `/etc/udev/rules.d/99-libftdi.rules`. You will need sudo access to create this file.
2. Put the following in the file:

```
SUBSYSTEMS=="usb", ATTRS{idVendor}=="0403", ATTRS{idProduct}=="6001",␣
↪GROUP="dialout", MODE="0660"
SUBSYSTEMS=="usb", ATTRS{idVendor}=="0403", ATTRS{idProduct}=="6010",␣
↪GROUP="dialout", MODE="0660"
SUBSYSTEMS=="usb", ATTRS{idVendor}=="0403", ATTRS{idProduct}=="6011",␣
↪GROUP="dialout", MODE="0660"
SUBSYSTEMS=="usb", ATTRS{idVendor}=="0403", ATTRS{idProduct}=="6014",␣
↪GROUP="dialout", MODE="0660"
SUBSYSTEMS=="usb", ATTRS{idVendor}=="0403", ATTRS{idProduct}=="6015",␣
↪GROUP="dialout", MODE="0660"
```

The list of USB product IDs above matches the default used by `pylibftdi`, but some FTDI devices may use other USB PIDs. You could try removing the match on `idProduct` altogether, just matching on the FTDI vendor ID as follows:

```
SUBSYSTEMS=="usb", ATTRS{idVendor}=="0403", GROUP="dialout", MODE="0660"
```

Or use `lsusb` or similar to determine the exact values to use (or try checking `dmesg` output on device insertion / removal). `udevadm monitor --environment` is also helpful, but note that the environment 'keys' it gives are different to the attributes (filenames within /sys/devices/...) which the ATTRS will match. Perhaps ENV{} matches work just as well, though I've only tried matching on ATTRS.

Note that changed udev rules files will be picked up automatically by the udev daemon, but will only be acted upon on device actions, so unplug/plug in the device to check whether you're latest rules iteration actually works :-)

Also note that the udev rules above assume that your user is in the 'dialout' group - if not, add it to your user with the following, though note that this will not apply immediately, not a full reboot may be needed on some systems:

```
sudo usermod -aG dialout $USER
```

See http://wiki.debian.org/udev for more on writing udev rules.

Arch Linux

The `libftdi` package (sensibly enough) provides the `libftdi` library:

```
$ sudo pacman -S libftdi
```

Similar udev rules to those above for Debian should be included (again in `/etc/udev/rules.d/99-libftdi.rules` or similar), though the GROUP directive should be changed to set the group to 'users':

```
SUBSYSTEMS=="usb", ATTRS{idVendor}=="0403", ATTRS{idProduct}=="6001",␣
↪GROUP="users", MODE="0660"
SUBSYSTEMS=="usb", ATTRS{idVendor}=="0403", ATTRS{idProduct}=="6010",␣
↪GROUP="users", MODE="0660"
(etc...)
```

1.3.4 Testing installation

Connect your device, and run the following (as a regular user):

```
$ python3 -m pylibftdi.examples.list_devices
```

If all goes well, the program should report information about each connected device. If no information is printed, but it is when run with `sudo`, a possibility is permissions problems - see the section under Linux above regarding `udev` rules.

If the above works correctly, then try the following:

```
$ python3 -m pylibftdi.examples.led_flash
```

Even without any LED connected, this should 'work' without any error - quit with Ctrl-C. Likely errors at this point are either permissions problems (e.g. udev rules not working), or not finding the device at all - although the earlier stage is likely to have failed if this were the case.

Feel free to contact me (@codedstructure on Twitter) if you have any issues with installation, though be aware I don't have much in the way of Windows systems to test.

1.4 Basic Usage

pylibftdi is a minimal Pythonic interface to FTDI devices using libftdi[7]. Rather than simply expose all the methods of the underlying library directly, it aims to provide a simpler API for the main use-cases of serial and parallel IO, while still allowing the use of the more advanced functions of the library.

[7] http://www.intra2net.com/en/developer/libftdi/

1.4.1 General

The primary interface is the `Device` class in the `pylibftdi` package; this gives serial access on relevant FTDI devices (e.g. the UM232R), providing a file-like interface (read, write). Baudrate is controlled with the `baudrate` property.

If a Device instance is created with `mode='t'` (text mode) then read() and write() can use the given `encoding` (defaulting to latin-1). This allows easier integration with passing unicode strings between devices.

Multiple devices are supported by passing the desired device serial number (as a string) in the `device_id` parameter - this is the first parameter in both Device() and BitBangDevice() constructors. Alternatively the device 'description' can be given, and an attempt will be made to match this if matching by serial number fails.

In the event that multiple devices (perhaps of identical type) have the same description and serial number, the `device_index` parameter may be given to open matching devices by numerical index; this defaults to zero, meaning the first matching device.

Examples

```
>>> from pylibftdi import Device
>>>
>>> with Device(mode='t') as dev:
...     dev.baudrate = 115200
...     dev.write('Hello World')
```

The pylibftdi.BitBangDevice wrapper provides access to the parallel IO mode of operation through the `port` and `direction` properties. These provide an 8 bit IO port including all the relevant bit operations to make things simple.

```
>>> from pylibftdi import BitBangDevice
>>>
>>> with BitBangDevice('FTE00P4L') as bb:
...     bb.direction = 0x0F  # four LSB are output(1), four MSB are␣
↪input(0)
...     bb.port |= 2         # set bit 1
...     bb.port &= 0xFE      # clear bit 0
```

There is support for a number of external devices and protocols, specifically for interfacing with HD44780 LCDs using the 4-bit interface.

1.5 Bit-bang mode

Bit-bang mode allows the programmer direct access (both read and write) to the state of the IO lines from a compatible FTDI device.

The interface provided by FTDI is intended to mirror the type of usage on a microcontroller, and is similar to the 'user port' on many old 8-bit computers such as the BBC Micro and Commodore 64.

The basic model is to have two 8 bit ports - one for data, and one for 'direction'. The data port maps each of the 8 bits to 8 independent IO signals, each of which can be configured separately as an 'input' or an 'output'.

In pylibftdi, the data port is given by the `port` attribute of a BitBangDevice instance, and the direction control is provided by the `direction` attribute. Both these attributes are implemented as Python properties, so no method calls are needed on them - simple read and write in Python-land converts to read and write in the physical world seen by the FTDI device.

The direction register maps to

where each bit maps to a separate digital signal,

1.5.1 Read-Modify-Write

Port vs Latch

Via the augmented assignment operations, pylibftdi `BitBangDevice` instances support read-modify-write operations, such as arithmetic (+= etc), bitwise (`&`=), and other logical operations such as shift (<<=)

Examples

```
>>> from pylibftdi import BitBangDevice
>>>
>>> with BitBangDevice('FTE00P4L') as bb:
...     bb.direction = 0x0F  # four LSB are output(1), four MSB are␣
↪input(0)
...     bb.port |= 2         # set bit 1
...     bb.port &= 0xFE      # clear bit 0


>>> with BitBangDevice() as bb:
...     bb.port = 1
...     while True:
...         # Rotate the value in bb.port
...         bb.port = ((bb.port << 1) | ((bb.port >> 8) & 1)) & 0xFF
...         time.sleep(1)
```

1.5.2 The *Bus* class

Dealing with bit masks and shifts gets messy quickly. Some languages such as C and C++ provide direct support for accessing bits - or series of consecutive bits - with bitfields. The `Bus` class provides the facility to provide a similar level of support to pylibftdi `BitBangDevice` classes.

As an example, consider an HD44780 LCD display. These have a data channel of either 4 or 8 bits, and a number of additional status lines - `rs` which acts as a register select pin - indicating whether a data byte is a command (0) or data (1), and `e` - clock enable.:

```
class LCD(object):
    """
    The UM232R/245R is wired to the LCD as follows:
       DB0..3 to LCD D4..D7 (pin 11..pin 14)
       DB6 to LCD 'RS' (pin 4)
       DB7 to LCD 'E' (pin 6)
    """
    data = Bus(0, 4)
```

(continues on next page)

(continued from previous page)

```
    rs = Bus(6)
    e = Bus(7)
```

1.6 Serial mode

The default mode of pylibftdi devices is to behave as a serial UART device, similar to the 'COM1' device found on older PCs. Nowadays most PCs operate with serial devices over USB-serial adapters, which may often include their own FTDI chips. To remain compatible with the RS232 standard however, these adapters will often include level-shifting circuitry which is of no benefit in communicating with other circuits operating at the 3.3 or 5 volt levels the FTDI hardware uses.

The default serial configuration is 9600 baud, 8 data bits, 1 stop bit and no parity (sometimes referred to as 8-N-1[8]). This is the default configuration of the old 'COM' devices back to the days of the original IBM PC and MS-DOS.

1.6.1 Setting line parameters

Changing line parameters other than the baudrate is supported via use of the underlying FTDI function calls.

1.6.2 The SerialDevice class

While the standard `Device` class supports standard `read` and `write` methods, as well as a `baudrate` property, further functionality is provided by the `SerialDevice` class, available either as a top-level import from `pylibftdi` or through the `serial_device` module. This subclasses `Device` and adds additional properties to access various control and handshake lines.

The following properties are available:

property	meaning	direction
`cts`	Clear To Send	Input
`rts`	Ready To Send	Output
`dsr`	Data Set Ready	Input
`dtr`	Data Transmit Ready	Output
`ri`	Ring Indicator	Input

Note that these lines are normally active-low, and `pylibftdi` makes no attempt to hide this from the user. It is impractical to try to 'undo' this inversion in any case, since it can be disabled in the EEPROM settings of the device. Just be aware if using these lines as GPIO that the electrical sense will be the opposite of the value read. The lines are intended to support handshaking rather than GPIO, so this is not normally an issue; if CTS is connected to RTS, then values written to RTS will be reflected in the value read from CTS.

[8] http://en.wikipedia.org/wiki/8-N-1

1.6.3 Subclassing *Device* - A MIDI device

To abstract application code from the details of any particular interface, it may be helpful to subclass the `Device` class, providing the required configuration in the `__init__` method to act in a certain way. For example, the MIDI[9] protocol used by electronic music devices is an asynchronous serial protocol operating at 31250 baud, and with the same 8-N-1 parameters which pylibftdi defaults to.

Creating a `MidiDevice` subclass of `Device` is straightforward:

```
class MidiDevice(Device):
    "subclass of pylibftdi.Device configured for MIDI"

    def __init__(self, *o, **k):
        Device.__init__(self, *o, **k)
        self.baudrate = 31250
```

Note it is important that the superclass `__init__` is called first; calling it on an uninitialised `Device` would fail, and even if it succeeded, the superclass `__init__` method resets `baudrate` to 9600 anyway to ensure a consistent setup for devices which may have been previously used with different parameters.

Use of the `MidiDevice` class is simple - as a pylibftdi Device instance, it provides a file-based API. Simply `read()` and `write()` the data to an instance of the class:

```
>>> m = MidiDevice()
>>> m.write('\x90\x80\x80')
>>> time.sleep(1)
>>> m.write('\x80\x00')
```

1.7 Advanced Usage

1.7.1 `libftdi` function access

Three attributes of `Device` instances are documented which allow direct access to the underlying `libftdi` functionality.

1. `fdll` - this is a reference to the loaded `libftdi` library, loaded via ctypes. This should be used with the normal ctypes protocols.
2. `ctx` - this is a reference to the context of the current device context. It is managed as a raw ctypes byte-string, so can be modified if required at the byte-level using appropriate `ctypes` methods.
3. `ftdi_fn` - a convenience function wrapper, this is the preferred method for accessing library functions for a specific device instance. This is a function forwarder to the local `fdll` attribute, but also wraps the device context and passes it as the first argument. In this way, using `device.ftdi_fn.ft_xyz` is more like the D2XX driver provided by FTDI, in which the device context is passed in at initialisation time and then the client no longer needs to care about it. A call to:

   ```
   >>> device.ftdi_fn.ft_xyz(1, 2, 3)
   ```

   is equivalent to the following:

[9] http://www.midi.org

```
>>> device.fdll.ft_xyz(ctypes.byref(device.ctx), 1, 2, 3)
```

but has the advantages of being shorter and not requiring ctypes to be in scope.

incorrect operations using any of these attributes of devices are liable to crash the Python interpreter

Examples

The following example shows opening a device in serial mode, switching temporarily to bit-bang mode, then back to serial and writing a string. Why this would be wanted is anyone's guess ;-)

```
>>> from pylibftdi import Device
>>>
>>> with Device() as dev:
>>>     dev.ftdi_fn.ftdi_set_bitmode(1, 0x01)
>>>     dev.write('\x00\x01\x00')
>>>     dev.ftdi_fn.ftdi_set_bitmode(0, 0x00)
>>>     dev.write('Hello World!!!')
```

The libftdi[10] documentation should be consulted in conjunction with the ctypes[11] reference for guidance on using these features.

1.8 pylibftdi questions

None of these are yet frequently asked, and perhaps they never will be... But they are still questions, and they relate to pylibftdi.

1.8.1 Using pylibftdi - General

Can I use pylibftdi with device XYZ?

If the device XYZ is (or uses as it's) an FTDI device, then possibly. A large number of devices *will* work, but won't be recognised due to the limited USB Vendor and Product IDs which pylibftdi checks for.

To see the vendor / product IDs which are supported, run the following:

```
>>> from pylibftdi import USB_VID_LIST, USB_PID_LIST
>>> print(', '.join(hex(pid) for pid in USB_VID_LIST))
0x403
>>> print(', '.join(hex(pid) for pid in USB_PID_LIST))
0x6001, 0x6010, 0x6011, 0x6014, 0x6015
```

If a FTDI device with a VID / PID not matching the above is required, then the device's values should be appended to the appropriate list after import:

[10] http://www.intra2net.com/en/developer/libftdi/documentation/
[11] http://docs.python.org/library/ctypes.html

```
>>> from pylibftdi import USB_PID_LIST, USB_VID_LIST, Device
>>> USB_PID_LIST.append(0x1234)
>>>
>>> dev = Device()  # will now recognise a device with PID 0x1234.
```

Which devices are recommended?

While I used to do a lot of soldering, I prefer the cleaner way of breadboarding nowadays. As such I can strongly recommend the FTDI DIP modules which plug into a breadboard nice and easy, can be self-powered from USB, and can be re-used for dozens of different projects.

I've used (and test against) the following, all of which have 0.1" pin spacing in two rows 0.5" or 0.6" apart, so will sit across the central divide of any breadboard:

UB232R a small 8 pin device with mini-USB port; serial and CBUS bit-bang.

UM245R a 24-pin device with parallel FIFO modes. Full-size USB type B socket.

UM232R a 24-pin device with serial and bit-bang modes. Full-size USB type B socket.

UM232H this contains a more modern FT232H device, and libftdi support is fairly recent (requires 0.20 or later). Supports USB 2.0 Hi-Speed mode though, and lots of interesting modes (I2C, SPI, JTAG...) which I've not looked at yet. Mini-USB socket.

Personally I'd go with the UM232R device for compatibility. It works great with both UART and bit-bang IO, which I target as the two main use-cases for pylibftdi. The UM232H is certainly feature-packed though, and I hope to support some of the more interesting modes in future.

1.8.2 Using pylibftdi - Programming

How do I set the baudrate?

In both serial and parallel mode, the internal baudrate generator (BRG) is set using the `baudrate` property of the `Device` instance. Reading this will show the current baudrate (which defaults to 9600); writing to it will attempt to set the BRG to that value.

On failure to set the baudrate, it will remain at its previous setting.

In parallel mode, the actual bytes-per-second rate of parallel data is 16x the programmed BRG value. This is an effect of the FTDI devices themselves, and is not hidden by pylibftdi.

How do I send unicode over a serial connection?

If a `Device` instance is created with `mode='t'`, then text-mode is activated. This is analogous to opening files; after all, the API is intentionally modelled on file objects whereever possible.

When text-mode is used, an encoding can be specified. The default is `latin-1` for the very practical reason that it is transparent to 8-bit binary data; by default a text-mode serial connection looks just like a binary mode one.

An alternative encoding can be used provided in the same constructor call used to instantiate the `Device` class, e.g.:

```
>>> dev = Device(mode='t', encoding='utf-8')
```

Read and write operations will then return / take unicode values.

Whether it is sensible to try and send unicode over a ftdi connection is a separate issue... At least consider doing codec operations at a higher level in your application.

How do I use multiple-interface devices?

Some FTDI devices have multiple interfaces, for example the FT2232H has 2 and the FT4232H has four. In terms of accessing them, they can be considered as independent devices; once a connection is established to one of them, it is isolated from the other interfaces.

To select which interface to use when opening a connection to a specific interface on a multiple-interface device, use the `interface_select` parameter of the Device (or BitBangDevice) class constructor. The value should be one of the following values. Symbolic constants are provided in the pylibftdi namespace.

`interface_select`	Meaning
INTERFACE_ANY (0)	Any interface
INTERFACE_A (1)	INTERFACE A
INTERFACE_B (2)	INTERFACE B
INTERFACE_C (3)	INTERFACE C
INTERFACE_D (4)	INTERFACE D

You should be able to open multiple `Devices` with different `interface_select` settings. *Thanks to Daniel Forer for testing multiple device support.*

What is the difference between the `port` and `latch` BitBangDevice properties?

latch reflects the current state of the output latch (i.e. the last value written to the port), while `port` reflects input states as well. Writing to either `port` or `latch` has an identical effect, so when pylibftdi is used only for output, there is no effective difference, and `port` is recommended for simplicity and consistency.

The place where it does make a difference is during read-modify-write operations. Consider the following:

```
>>> dev = BitBangDevice()   # 1
>>> dev.direction = 0x81    # 2   # set bits 0 and 7 are output
>>> dev.port = 0            # 3
>>> for _ in range(255):    # 4
>>>     dev.port += 1        # 5   # read-modify-write operation
```

In this (admittedly contrived!) scenario, if one of the input lines D1..D6 were held low, then they would cause the counter to effectively 'stop'. The `+= 1` operation would never actually set the bit as required (because it is an input at 0), and the highest output bit would never get set.

Using `dev.latch` in lines 3 and 5 above would resolve this, as the read-modify-write operation on line 5 is simply working on the in-memory latch value, rather than reading the inputs, and it would simply count up from 0 to 255 in steps of one, writing the value to the device (which would be ignored in the case of input lines).

Similar concepts exist in many microcontrollers, for example see http://stackoverflow.com/a/2623498 for a possibly better explanation, though in a slightly different context :)

If you aren't using read-modify-write operations (e.g. augmented assignment), or you have a direction on the port of either ALL_INPUTS (0) or ALL_OUTPUTS (1), then just ignore this section and use `port` :)

What is the purpose of the `chunk_size` parameter?

While libftdi is performing I/O to the device, it is not really running Python code at all, but C library code via ctypes. If there is a significant amount of data, especially at low baud-rates, this can be a significant delay during which no Python bytecode is executed. The most obvious result of this is that no signals are delivered to the Python process during this time, and interrupt signals (Ctrl-C) will be ignored.

Try the following:

```
>>> dev = Device()
>>> dev.baudrate = 120  # nice and slow!
>>> dev.write('helloworld' * 1000)
```

This should take approximately 10 seconds prior to returning, and crucially, Ctrl-C interruptions will be deferred for all that time. By setting `chunk_size` on the device (which may be set either as a keyword parameter during `Device` instantiation, or at a later point as an attribute of the `Device` instance), the I/O operations are performed in chunks of at most the specified number of bytes. Setting it to 0, the default value, disables this chunking.

Repeat the above command but prior to the write operation, set `dev.chunk_size = 10`. A Ctrl-C interruption should now kick-in almost instantly. There is a performance trade-off however; if using `chunk_size` is required, set it as high as is reasonable for your application.

1.8.3 Using pylibftdi - Interfacing

How do I control an LED?

pylibftdi devices generally have sufficient output current to sink or source the 10mA or so which a low(ish) current LED will need. A series resistor is essential to protect both the LED and the FTDI device itself; a value between 220 and 470 ohms should be sufficient depending on required brightness / LED efficiency.

How do I control a higher current device?

FTDI devices will typically provide a few tens of milli-amps, but beyond that things either just won't work, or the device could be damaged. For medium current operation, a standard bipolar transistor switch will suffice; for larger loads a MOSFET or relay should be used. (Note a relay will require a low-power transistor switch anyway). Search online for something like 'mosfet logic switch' or 'transistor relay switch' for more details.

What is the state of an unconnected input pin?

This depends on the device and the EEPROM configuration values. Most devices will have weak (typ. 200Kohm) pull-ups on input pins, so there is no harm leaving them floating. Consult the datasheet for

your device for definitive information, but you can always just leave an (unconnected) device and read it's pins when set as inputs; chances are they will read 255 / 0xFF:

```
>>> dev = BitBangDevice(direction=0)
>>> dev.port
255
```

While not recommended for anything serious, this does allow the possibility of reading a input switch state by simply connecting a switch between an input pin and ground (possibly with a low value - e.g. 100 ohm - series resistor to prevent accidents should it be set to an output and set high...). Note that with a normal push-to-make switch, the value will read '1' when the switch is not pressed; pressing it will set the input line value to '0'.

1.9 pylibftdi troubleshooting

Once up-and-running, pylibftdi is designed to be very simple, but sometimes getting it working in the first place can be more difficult.

1.9.1 Error messages

FtdiError: unable to claim usb device. Make sure the default FTDI driver is not in use (-5)

This indicates a conflict with FTDI's own drivers, and is (as far as I know) mainly a problem on Mac OS X, where they can be disabled (until reboot) by unloading the appropriate kernel module.

MacOS (Mavericks and later)

Starting with OS X Mavericks, OS X includes kernel drivers which will reserve the FTDI device by default. In addition, the FTDI-provided VCP driver will claim the device by default. These need unloading before *libftdi* will be able' to communicate with the device:

```
sudo kextunload -bundle-id com.apple.driver.AppleUSBFTDI
sudo kextunload -bundle-id com.FTDI.driver.FTDIUSBSerialDriver
```

Similarly to reload them:

```
sudo kextload -bundle-id com.apple.driver.AppleUSBFTDI
sudo kextload -bundle-id com.FTDI.driver.FTDIUSBSerialDriver
```

Earlier versions of `pylibftdi` (prior to 0.18.0) included scripts for MacOS which unloaded / reloaded these drivers, but these complicated cross-platform packaging so have been removed. If you are on using MacOS with programs which need these drivers on a frequent basis (such as the Arduino IDE when using older FTDI-based Arduino boards), consider implementing these yourself, along the lines of the following (which assumes ~/bin is in your path):

```
cat << EOF > /usr/local/bin/ftdi_osx_driver_unload
sudo kextunload -bundle-id com.apple.driver.AppleUSBFTDI
sudo kextunload -bundle-id com.FTDI.driver.FTDIUSBSerialDriver
EOF
```

(continues on next page)

(continued from previous page)

```
cat << EOF > /usr/local/bin/ftdi_osx_driver_reload
sudo kextload -bundle-id com.apple.driver.AppleUSBFTDI
sudo kextload -bundle-id com.FTDI.driver.FTDIUSBSerialDriver
EOF

chmod +x /usr/local/bin/ftdi_osx_driver_*
```

OS X Mountain Lion and earlier

Whereas Mavericks includes an FTDI driver directly, earlier versions of OS X did not, and if this issue occurred it would typically as a result of installing some other program - for example the Arduino IDE.

As a result, the kernel module may have different names, but *FTDIUSBSerialDriver.kext* is the usual culprit. Unload the kernel driver as follows:

```
sudo kextunload /System/Library/Extensions/FTDIUSBSerialDriver.kext
```

To reload the kernel driver, do the following:

```
sudo kextload /System/Library/Extensions/FTDIUSBSerialDriver.kext
```

If you aren't using whatever program might have installed it, the driver could be permanently removed (to prevent the need to continually unload it), but this is dangerous:

```
sudo rm /System/Library/Extensions/FTDIUSBSerialDriver.kext
```

1.9.2 Diagnosis

Getting a list of USB devices

Mac OS X

Start 'System Information', then select Hardware > USB, and look for your device. On the command line, `system_profiler SPUSBDataType` can be used. In the following example I've piped it into `grep -C 7 FTDI`, to print 7 lines either side of a match on the string 'FTDI':

```
ben$ system_profiler SPUSBDataType | grep -C 7 FTDI
        UM232H:

          Product ID: 0x6014
          Vendor ID: 0x0403  (Future Technology Devices International␣
↪Limited)
          Version: 9.00
          Serial Number: FTUBIOWF
          Speed: Up to 480 Mb/sec
          Manufacturer: FTDI
          Location ID: 0x24710000 / 7
          Current Available (mA): 500
          Current Required (mA): 90
```

(continues on next page)

(continued from previous page)

```
        USB Reader:

          Product ID: 0x4082
```

Linux

Use `lsusb`. Example from my laptop:

```
ben@ben-laptop:~$ lsusb
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 003 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 004 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 005 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 006 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 007 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 008 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 008 Device 011: ID 0a5c:217f Broadcom Corp. Bluetooth Controller
Bus 002 Device 009: ID 17ef:481d Lenovo
Bus 002 Device 016: ID 0403:6014 Future Technology Devices International,␣
↪Ltd FT232H Single HS USB-UART/FIFO IC
```

1.9.3 Where did my ttyUSB devices go?

When a *pylibftdi.Device()* is opened, any kernel device which was previously present will become unavailable. On Linux for example, a serial-capable FTDI device will (via the *ftdi_sio* driver) create a device node such as */dev/ttyUSB0* (or ttyUSB1,2,3 etc). This device allows use of the FTDI device as a simple file in the Linux filesystem which can be read and written. Various programs such as the Arduino IDE (at least when communicating with some board variants) and libraries such as *PySerial* will use this device. Once libftdi opens a device, the corresponding entry in /dev/ will disappear. Prior to *pylibftdi* version 0.16, the simplest way to get the device node to reappear would be to unplug and replug the USB device itself. Starting from 0.16, this should no longer be necessary as the kernel driver (which exports */dev/ttyUSB...*) is reattached when the *pylibftdi* device is closed. This behaviour can be controlled by the *auto_detach* argument (which is defaulted to *True*) to the *Device* class; setting it to *False* reverts to the old behaviour.

Note that on recent OS X, libftdi doesn't 'steal' the device, but instead refuses to open it. The kernel devices can be seen as */dev/tty.usbserial-xxxxxxxx*, where *xxxxxxxx* is the device serial number. FTDI's Application Note AN134[12] details this further (see section 'Using Apple-provided VCP or D2XX with OS X 10.9 & 10.10'). See the section above under Installation for further details on resolving this.

1.9.4 Gathering information

Starting with pylibftdi version 0.15, an example script to gather system information is included, which will help in any diagnosis required.

Run the following:

[12] http://www.ftdichip.com/Support/Documents/AppNotes/AN_134_FTDI_Drivers_Installation_Guide_for_MAC_OSX.pdf

```
python3 -m pylibftdi.examples.info
```

this will output a range of information related to the versions of libftdi libusb in use, as well as the system platform and Python version, for example:

```
pylibftdi version     : 0.18.0
libftdi version       : libftdi_version(major=1, minor=4, micro=0, version_
↪str='1.4', snapshot_str='unknown')
libftdi library name  : libftdi1.so.2
libusb version        : libusb_version(major=1, minor=0, micro=22,␣
↪nano=11312, rc='', describe='http://libusb.info')
libusb library name   : libusb-1.0.so.0
Python version        : 3.7.3
OS platform           : Linux-5.0.0-32-generic-x86_64-with-Ubuntu-19.04-
↪disco
```

1.10 Developing pylibftdi

1.10.1 How do I checkout and use the latest development version?

pylibftdi is currently developed on GitHub, though started out as a Mercurial repository on bitbucket.org. There may still be references to old bitbucket issues in the docs.

pylibftdi is developed using poetry[13], and a Dockerfile plus Makefile make use development tasks straightforward. In any case, start with a local clone of the repository:

```
$ git clone https://github.com/codedstructure/pylibftdi
$ cd pylibftdi
```

There are then two main approaches, though pick and mix the different elements to suit:

poetry and docker If *make* and *docker* are available in your environment, the easiest way to do development may be to simply run *make shell*. This creates an Ubuntu-based docker environment with *libftdi*, *poetry*, and other requirements pre-installed, and drops into a shell where the current *pylibftdi* code is installed.

make on its own will run through all the unittests and linting available for *pylibftdi*, and is a useful check to make sure things haven't been broken.

The downside of running in a docker container is that USB support to actual FTDI devices may be lacking...

editable install with pip This assumes that the *venv* and *pip* packages are installed; on some (e.g. Ubuntu) Linux environments, these may need installing as OS packages. Once installed, perform an 'editable' install as follows:

```
.../pylibftdi$ python3 -m venv env
.../pylibftdi$ source env/bin/activate
(env) .../pylibftdi$ python3 -m pip install -e .
```

Note this also creates a virtual environment within the project directory; see here[14]

[13] https://python-poetry.org/
[14] https://packaging.python.org/guides/installing-using-pip-and-virtual-environments/

1.10.2 How do I run the tests?

From the root directory of a cloned pylibftdi repository, run the following:

```
(env) .../pylibftdi$ python3 -m unittest discover
.....................................
----------------------------------------------------------------------
Ran 37 tests in 0.038s

OK
```

Note that other test runners (such as *pytest*) will also run the tests and may be easier to extend.

1.10.3 How can I determine and select the underlying libftdi library?

Since pylibftdi 0.12, the Driver exposes `libftdi_version()` and `libusb_version()` methods, which return a tuple whose first three entries correspond to major, minor, and micro versions of the libftdi driver being used.

Note there are two major versions of *libftdi* - libftdi1 can coexist with the earlier 0.x versions - it is now possible to select which library to load when instantiating the Driver. Note on at least Ubuntu Linux, the *libftdi1 OS package* actually refers to *libftdi 0.20* (or similar), whereas *libftdi1-2* refers to the more recent 1.x release (currently 1.5):

```
Python 3.10.6 (main, May 29 2023, 11:10:38) [GCC 11.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from pylibftdi import Driver
>>> Driver().libftdi_version()
libftdi_version(major=1, minor=5, micro=0, version_str='1.5', snapshot_str=
↪'unknown')
>>> Driver("ftdi1").libftdi_version()
libftdi_version(major=1, minor=5, micro=0, version_str='1.5', snapshot_str=
↪'unknown')
>>> Driver("ftdi").libftdi_version()
libftdi_version(major=0, minor=0, micro=0, version_str='< 1.0 - no ftdi_
↪get_library_version()', snapshot_str='unknown')
```

If both are installed, `pylibftdi` prefers libftdi1 (e.g. libftdi 1.5) over libftdi (e.g. 0.20). Since different OSs require different parameters to be given to find a library, the default search list given to ctypes.util.find_library is defined by the *Driver._lib_search* attribute, and this may be updated as appropriate. By default it is as follows:

```
_lib_search = {
    "libftdi": ["ftdi1", "libftdi1", "ftdi", "libftdi"],
    "libusb": ["usb-1.0", "libusb-1.0"],
}
```

This covers Windows (which requires the 'lib' prefix), Linux (which requires its absence), and Mac OS X, which is happy with either.

1.11 pylibftdi Package

1.11.1 pylibftdi Package

1.11.2 _base Module

1.11.3 device Module

1.11.4 driver Module

1.11.5 bitbang Module

1.11.6 serial_device Module

1.11.7 util Module

1.11.8 Subpackages

examples Package

examples Package

bit_server Module

lcd Module

led_flash Module

list_devices Module

magic_candle Module

midi_output Module

pin_read Module

serial_loopback Module

info Module

CHAPTER 2

Indices and tables

- genindex
- modindex
- search