Facedancer

Great Scott Gadget

May 07, 2024

USER DOCUMENTATION

CHAPTER

ONE

GETTING STARTED WITH FACEDANCER

1.1 Install the Facedancer library

You can install the Facedancer library from the Python Package Index (PyPI), a release archive or directly from source.

1.1.1 Install From PyPI

You can use the pip tool to install the Facedancer library from PyPI using the following command:

```
pip install facedancer
```

For more information on installing Python packages from PyPI please refer to the "Installing Packages" section of the Python Packaging User Guide.

1.1.2 Install From Source

```
git clone https://github.com/greatscottgadgets/facedancer.git
cd facedancer/
```

Once you have the source code downloaded you can install the Facedancer library with:

```
pip install .
```

1.2 Run a Facedancer example

Create a new Python file called *rubber-ducky.py* with the following content:

```
import asyncio
import logging

from facedancer import main
from facedancer.devices.keyboard     import USBKeyboardDevice
from facedancer.classes.hid.keyboard import KeyboardModifiers

device = USBKeyboardDevice()

async def type_letters():
```

(continues on next page)

(continued from previous page)

```
    # Wait for device to connect
    await asyncio.sleep(2)

    # Type a string with the device
    await device.type_string("echo hello, facedancer\n")

main(device, type_letters())
```

Open a terminal and run:

```
python ./rubber-ducky.py
```

CHAPTER

TWO

LIBRARY OVERVIEW

The Facedancer library may be somewhat overwhelming at first but the modules can be broken down into a number of clearly delineated categories:

2.1 Core USB Device Model

These packages contain the functionality used to define devices and their organisation closely mirrors the hierarchical USB device model:

```
+--------------------------------+
| USB Device                     |
|   - Device Descriptor          |
|   - Configuration Descriptor   |
|     - Interface Descriptor     |     +-----------------------------------+
|       - Endpoint Descriptor    |     | Host                              |
|         - Request Handler      | --> |   - Function                      |
|       - Endpoint Descriptor    |     |                                   |
|         - Request Handler      | <-- |   - Function                      |
|   - Control Interface          |     |                                   |
|     - Request Handlers         | <-> |   - Enumeration, Status, Command  |
+--------------------------------+     +-----------------------------------+

(simplified diagram for didactic purposes, not drawn to scale)
```

- *facedancer.device*
  – *USBDevice* is the device root. It is responsible for managing the device's descriptors and marshalling host requests.
- *facedancer.configuration*
  – *USBConfiguration* is responsible for managing the device's configuration descriptor(s).
- *facedancer.interface*
  – *USBInterface* is responsible for managing the device's interface descriptor(s).
- *facedancer.endpoint*
  – *USBEndpoint* is responsible for managing the device's endpoints.
- *facedancer.request*
  – *USBControlRequest* is responsible for managing USB control transfers.

In addition to the core device model there are also two modules containing support functionality:

- *facedancer.descriptor*
  – contains functionality for working with USB descriptors.

- *facedancer.magic*
  – contains functionality for Facedancer's declarative device definition syntax.

2.2 Device Emulation Support

These modules contain a small selection of example USB device classes and device emulations.

- *facedancer.classes*
- *facedancer.devices*

2.3 USB Proxy

These modules contain the USB Proxy implementation.

- *facedancer.proxy*
  – contains the *USBProxyDevice* implementation.
- *facedancer.filters*
  – contains a selection of filters to intercept, view or modify proxied USB transfers.

2.4 Facedancer Board Backends

Contains backend implementations for the various supported Facedancer boards.

- *facedancer.backends*

2.5 Supporting Functionality

- *facedancer.core*
  – the Facedancer scheduler and execution core.
- *facedancer.errors*
  – an error type, there should probably be more.
- *facedancer.types*
  – various type definitions and constants.
- *facedancer.logging*
  – logging boilerplate.

CHAPTER

THREE

USING FACEDANCER

3.1 Introduction

Facedancer allows you to easily define emulations using a simple declarative DSL that mirrors the hierarchical structure of the abstract USB device model.

Let's look at a simple example that defines a USB device with two endpoints and a control interface:

```
import logging

from facedancer import *
from facedancer import main

@use_inner_classes_automatically
class MyDevice(USBDevice):
    product_string      : str = "Example USB Device"
    manufacturer_string : str = "Facedancer"
    vendor_id           : int = 0x1209
    product_id          : int = 0x0001
    device_speed        : DeviceSpeed = DeviceSpeed.FULL

    class MyConfiguration(USBConfiguration):

        class MyInterface(USBInterface):

            class MyInEndpoint(USBEndpoint):
                number          : int          = 1
                direction       : USBDirection = USBDirection.IN
                max_packet_size : int          = 64

                def handle_data_requested(self: USBEndpoint):
                    logging.info("handle_data_requested")
                    self.send(b"device on bulk endpoint")

            class MyOutEndpoint(USBEndpoint):
                number          : int          = 1
                direction       : USBDirection = USBDirection.OUT
                max_packet_size : int          = 64

                def handle_data_received(self: USBEndpoint, data):
                    logging.info(f"device received {data} on bulk endpoint")
```

(continues on next page)

(continued from previous page)

```
    @vendor_request_handler(number=1, direction=USBDirection.IN)
    @to_device
    def my_in_vendor_request_handler(self: USBDevice, request: USBControlRequest):
        logging.info("my_in_vendor_request_handler")
        request.reply(b"device on control endpoint")

    @vendor_request_handler(number=2, direction=USBDirection.OUT)
    @to_device
    def my_out_vendor_request_handler(self: USBDevice, request: USBControlRequest):
        logging.info(f"device received {request.index} {request.value} {bytes(request.
↪data)} on control endpoint")
        request.ack()


main(MyDevice)
```

3.2 Device Descriptor

The entry-point for most Facedancer emulations is the *USBDevice* class which maintains the configuration as well as the transfer handling implementation of the device under emulation.

Note: In some cases you may want to use the *USBBaseDevice* class if you'd like to provide your own implementation of the standard request handlers.

See, for example, *USBProxyDevice*.

Starting with the initial class declaration we can define our device as:

```
from facedancer import *

@use_inner_classes_automatically
class MyDevice(USBDevice):
    product_string      : str = "Example USB Device"
    manufacturer_string : str = "Facedancer"
    vendor_id           : int = 0x1209 # https://pid.codes/1209/
    product_id          : int = 0x0001
```

We start by importing the Facedancer library and declaring a class *MyDevice* derived from *USBDevice*.

We also annotate our class with the *@use_inner_classes_automatically* decorator which allows us to use a declarative style when including our devices configuration, interface and endpoints. It's *magic*!

Finally, we fill in some basic fields Facedancer will use to populate the device descriptor: `product_string`, `manufacturer_string`, `vendor_id` and `product_id`.

Note: You can find a full list of supported fields in the *USBDevice* API documentation.

3.3 Configuration Descriptor

Once we have provided Facedancer with the basic information it needs to build a device descriptor we can move on to declare and define our device's configuration descriptor.

Most devices consist of a single configuration managed by the *USBConfiguration* class containing at least one *USBInterface* class containing zero or more *USBEndpoint* class.

Here we define a configuration with a single interface containing two endpoints. The first endpoint has direction *IN* and will be responsible for responding to data requests from the host. The second endpoint has direction *OUT* and will be responsible for receiving data from the host.

```
...
class MyDevice(USBDevice):
    ...

    class MyConfiguration(USBConfiguration):
        class MyInterface(USBInterface):
            class MyInEndpoint(USBEndpoint):
                number    : int          = 1
                direction : USBDirection = USBDirection.IN
            class MyOutEndpoint(USBEndpoint):
                number    : int          = 1
                direction : USBDirection = USBDirection.OUT
```

We've now provided enough information in our emulation for it to be successfully enumerated and recognized by the host but there is still one thing missing!

3.4 Request Handlers

For our device to actually do something we also need a way to:

- Respond to a request for data from the host.
- Receive data sent by the host.

Note: USB is a polled protocol where the host always initiates all transactions. Data will only ever be sent from the device if the host has first requested it from the device.

The Facedancer *facedancer.endpoint* and *facedancer.request* modules provides the functionality for responding to requests on the device's endpoints and the control interface. (All USB devices support a control endpoint – usually endpoint zero.)

3.4.1 Endpoint Request Handlers

Endpoint request handlers are usually either class-specific or vendor-defined and can be declared inside the device's endpoint declaration.

Here we will define two simple handlers for each endpoint.

For our IN endpoint we will reply to any data request from the host with a fixed message and for our OUT endpoint we will just print the received data to the terminal.

```
...
class MyDevice(USBDevice):
    ...

    class MyConfiguration(USBConfiguration):
        class MyInterface(USBInterface):
            class MyInEndpoint(USBEndpoint):
                number    : int          = 1
                direction : USBDirection = USBDirection.IN

                # called when the host requested data from the device on endpoint 0x81
                def handle_data_requested(self: USBEndpoint):
                    self.send(b"device sent response on bulk endpoint", blocking=True)

            class MyOutEndpoint(USBEndpoint):
                number    : int          = 1
                direction : USBDirection = USBDirection.OUT

                # called when the host sent data to the device on endpoint 0x01
                def handle_data_received(self: USBEndpoint, data):
                    logging.info(f"device received '{data}' on bulk endpoint")
```

For more information on supported endpoint operations and fields see the *USBEndpoint* documentation.

3.4.2 Control Request Handlers

Control Requests are typically used for command and status operations. While Facedancer will take care of responding to standard control requests used for device enumeration you may also want to implement custom vendor requests or even override standard control request handling.

To this end, Facedancer provides two sets of decorators to be used when defining a device's control interface:

The first set of decorators allows you to specify the type of control request to be handled:

- *@control_request_handler*
- *@standard_request_handler*
- *@vendor_request_handler*
- *@class_request_handler*
- *@reserved_request_handler*

The second set defines the target for the control request:

- *@to_device*
- *@to_this_endpoint*

- *@to_any_endpoint*
- *@to_this_interface*
- *@to_any_interface*
- *@to_other*

For instance, to define some vendor request handlers you can do:

```
...
class MyDevice(USBDevice):
    ...
    class MyConfiguration(USBConfiguration):
    ...

    @vendor_request_handler(request_number=1, direction=USBDirection.IN)
    @to_device
    def my_vendor_request_handler(self: USBDevice, request: USBControlRequest):
        request.reply(b"device sent response on control endpoint")

    @vendor_request_handler(request_number=2, direction=USBDirection.OUT)
    @to_device
    def my_other_vendor_request_handler(self: USBDevice, request: USBControlRequest):
        logging.info(f"device received '{request.index}' '{request.value}' '{request.
↪data}' on control endpoint")

        # acknowledge the request
        request.ack()
```

More information on the `request` parameter can be found in the *USBControlRequest* documentation.

3.5 Testing The Emulation

We now have a full USB device emulation that will enumerate and respond to requests from the host.

Give it a try!

```
import logging

def main():
    import asyncio
    import usb1

    VENDOR_REQUEST    = 0x65
    MAX_TRANSFER_SIZE = 64

    with usb1.USBContext() as context:
        #logging.info("Host: waiting for device to connect")
        #await asyncio.sleep(1)

        device_handle = context.openByVendorIDAndProductID(0x1209, 0x0001)
        if device_handle is None:
            raise Exception("device not found")
```

(continues on next page)

(continued from previous page)

```
        device_handle.claimInterface(0)

        # test IN endpoint
        logging.info("Testing bulk IN endpoint")
        response = device_handle.bulkRead(
            endpoint = 0x81,
            length   = MAX_TRANSFER_SIZE,
            timeout  = 1000,
        )
        logging.info(f"[host] received '{response}' from bulk endpoint")
        print("")

        # test OUT endpoint
        logging.info("Testing bulk OUT endpoint")
        response = device_handle.bulkWrite(
            endpoint = 0x01,
            data     = b"host say oh hai on bulk endpoint",
            timeout  = 1000,
        )
        print(f"sent {response} bytes\n")

        # test IN vendor request handler
        logging.info("Testing IN control transfer")
        response = device_handle.controlRead(
            request_type = usb1.TYPE_VENDOR | usb1.RECIPIENT_DEVICE,
            request      = 1,
            index        = 2,
            value        = 3,
            length       = MAX_TRANSFER_SIZE,
            timeout      = 1000,
        )
        logging.info(f"[host] received '{response}' from control endpoint")
        print("")

        # test OUT vendor request handler
        logging.info("Testing OUT control transfer")
        response = device_handle.controlWrite(
            request_type = usb1.TYPE_VENDOR | usb1.RECIPIENT_DEVICE,
            request      = 2,
            index        = 3,
            value        = 4,
            data         = b"host say oh hai on control endpoint",
            timeout      = 1000,
        )
        print(f"sent {response} bytes\n")


if __name__ == "__main__":
    logging.getLogger().setLevel(logging.DEBUG)
    main()
```

3.6 Suggestion Engine

Facedancer provides a suggestion engine that can help when trying to map an undocumented device's control interface.

It works by monitoring the control requests from the host and tracking any which are not supported by your emulation.

You can enable it by passing the *–suggest* flag when running an emulation:

```
python ./emulation.py --suggest
```

When you exit the emulation it can then suggest the handler functions you still need to implement in order to support the emulated device's control interface:

```
Automatic Suggestions
---------------------
These suggestions are based on simple observed behavior;
not all of these suggestions may be useful / desirable.

Request handler code:

@vendor_request_handler(number=1, direction=USBDirection.IN)
@to_device
def handle_control_request_1(self, request):
    # Most recent request was for 64B of data.
    # Replace me with your handler.
    request.stall()
```

3.7 Annotated template

The Facedancer repository contains an annotated template which provides an excellent reference source when building your own devices:

```
import logging

from facedancer         import main
from facedancer         import *
from facedancer.classes import USBDeviceClass

@use_inner_classes_automatically
class TemplateDevice(USBDevice):
    """ This class is meant to act as a template to help you get acquainted with␣
↪FaceDancer."""

    #
    # Core 'dataclass' definitions.
    # These define the basic way that a FaceDancer device advertises itself to the host.
    #
    # Every one of these is optional. The defaults are relatively sane, so you can mostly
    # ignore these unless you want to change them! See the other examples for more␣
↪minimal
    # data definitions.
    #
```

(continues on next page)

(continued from previous page)

```
    # The USB device class, subclass, and protocol for the given device.
    # Often, we'll leave these all set to 0, which means the actual class is read
    # from the interface.
    #
    # Note that we _need_ the type annotations on these. Without them, Python doesn't
    # consider them valid dataclass members, and ignores them. (This is a detail of␣
↪python3.7+
    # dataclasses.)
    #
    device_class            : int  = 0
    device_subclass         : int  = 0
    protocol_revision_number : int  = 0

    # The maximum packet size on EP0. For most devices, the default value of 64 is fine.
    max_packet_size_ep0     : int  = 64

    # The vendor ID and product ID that we want to give our device.
    vendor_id               : int  = 0x610b
    product_id              : int  = 0x4653

    # The string descriptors we'll provide for our device.
    # Note that these should be Python strings, and _not_ bytes.
    manufacturer_string     : str  = "FaceDancer"
    product_string          : str  = "Generic USB Device"
    serial_number_string    : str  = "S/N 3420E"

    # This tuple is a list of languages we're choosing to support.
    # This gives us an opportunity to provide strings in various languages.
    # We don't typically use this; so we can leave this set to a language of
    # your choice.
    supported_languages     : tuple = (LanguageIDs.ENGLISH_US,)

    # The revision of the device hardware. This doesn't matter to the USB specification,
    # but it's sometimes read by drivers.
    device_revision         : int  = 0

    # The revision of the USB specification that this device adheres to.
    # Typically, you'll leave this at '2'.
    usb_spec_version        : int  = 0x0002


    #
    # We'll define a single configuration on our device. To be compliant,
    # every device needs at least a configuration and an interface.
    #
    # Note that we don't need to do anything special to have this be used.
    # As long as we're using the @use_inner_classes_automatically decorator,
    # this configuration will automatically be instantiated and used.
    #
    class TemplateConfiguration(USBConfiguration):

```

(continues on next page)

(continued from previous page)

```
        #
        # Configuration fields.
        #
        # Again, all of these are optional; and the default values
        # are sane and useful.
        #

        # Configuration number. Every configuration should have a unique
        # number, which should count up from one. Note that a configuration
        # shouldn't have a number of 0, as that's USB for "unconfigured".
        configuration_number : int            = 1

        # A simple, optional descriptive name for the configuration. If provided,
        # this is referenced in the configuration's descriptor.
        configuration_string : str            = None

        # This setting is set to true if the device can run without bus power,
        # or false if it pulls its power from the USB bus.
        self_powered         : bool           = False

        # This setting is set to true if the device can ask that the host
        # wake it up from sleep. If set to true, the host may choose to
        # leave power on to the device when the host is suspended.
        supports_remote_wakeup : bool         = True

        # The maximum power the device will use in this configuration, in mA.
        # Typically, most devices will request 500mA, the maximum allowed.
        max_power              : int          = 500


        class TemplateInterface(USBInterface):

            #
            # Interface fields.
            # Again, all optional and with useful defaults.
            #

            # The interface index. Each interface should have a unique index,
            # starting from 0.
            number                : int = 0

            # The information about the USB class implemented by this interface.
            # This is the place where you'd specify if this is e.g. a HID device.
            class_number          : int = USBDeviceClass.VENDOR_SPECIFIC
            subclass_number       : int = 0
            protocol_number       : int = 0

            # A short description of the interface. Optional and typically only␣
↪informational.
            interface_string      : str = None


```

(continues on next page)

(continued from previous page)

```
            #
            # Here's where we define any endpoints we want to add to the device.
            # These behave essentially the same way as the above.
            #
            class TemplateInEndpoint(USBEndpoint):

                #
                # Endpoints are unique in that they have two _required_
                # properties -- their number and direction.
                #
                # Together, these two fields form the endpoint's address.
                number              : int                   = 1
                direction           : USBDirection          = USBDirection.IN

                #
                # The remainder of the fields are optional and have useful defaults.
                #

                # The transfer type selects how data will be transferred over the␣
↪endpoints.
                # The currently supported types are BULK and INTERRUPT.
                transfer_type       : USBTransferType       = USBTransferType.BULK

                # The maximum packet size determines how large packets are allowed to be.
                # For a full speed device, a max-size value of 64 is typical.
                max_packet_size     : int = 64

                # For interrupt endpoints, the interval specifies how often the host␣
↪should
                # poll the endpoint, in milliseconds. 10ms is a typical value.
                interval            : int = 0


                #
                # Let's add an event handler. This one is called whenever the host
                # wants to read data from the device.
                #
                def handle_data_requested(self):

                    # We can reply to this request using the .send() method on this
                    # endpoint, like so:
                    self.send(b"Hello!")

                    # We can also get our parent interface using .parent;
                    # or a reference to our device using .get_device().


            class TemplateOutEndpoint(USBEndpoint):
                #
                # We'll use a more typical set of properties for our OUT endpoint.
                #
                number              : int                   = 1
```

(continues on next page)

(continued from previous page)

```
                direction           : USBDirection          = USBDirection.OUT


                #
                # We'll also demonstrate use of another event handler.
                # This one is called whenever data is sent to this endpoint.
                #
                def handle_data_received(self, data):
                    logging.info(f"Received data: {data}")


    #
    # Any of our components can use callback functions -- not just our endpoints!
    # The callback names are the same no matter where we use them.
    #
    def handle_data_received(self, endpoint, data):

        #
        # When using a callback on something other than an endpoint, our function's
        # signature is slightly different -- it takes the relevant endpoint as an
        # argument, as well.
        #

        # We'll delegate this back to the core handler, here, so it propagates to our␣
↪subordinate
        # endpoints -- but we don't have to! If we wanted to, we could call functions on␣
↪the
        # endpoint itself. This is especially useful if we're hooking handle_data_
↪requested(),
        # where we can use endpoint.send() to provide the relevant data.
        super().handle_data_received(endpoint, data)

        # Note that non-endpoints have a get_endpoint() method, which you can use to get␣
↪references
        # to endpoints by their endpoint numbers / directions. This is useful if you␣
↪want to
        # send something on another endpoint in response to data received.
        #
        # The device also has a .send() method, which accepts an endpoint number and the␣
↪data to
        # be sent. This is equivalent to calling .send() on the relevant endpoint.


    #
    # We can very, very easily add request handlers to our devices.
    #
    @vendor_request_handler(number=12)
    def handle_my_request(self, request):

        #
        # By decorating this function with "vendor_request_handler", we've ensured this
        # function is called to handle vendor request 12. We can also add other␣
```

(continues on next page)

(continued from previous page)

```
↪arguments to
        # the vendor_request_handler function -- it'll accept a keyword argument for every
        # property on the request. If you provide these, the handler will only be called
        # if the request matches the relevant constraint.
        #
        # For example, @vendor_request_handler(number=14, direction=USBDirection.IN,␣
↪index_low=3)
        # means the decorated function is only called to handle vendor request 14 for IN␣
↪requests
        # where the low byte of the index is 3.
        #
        # Other handler decorators exist -- like "class_request_handler" or "standard_
↪request_handler"
        #

        # Replying to an IN request is easy -- you just provide the reply data using␣
↪request.reply().
        request.reply(b"Hello, there!")


    @vendor_request_handler(number=1, direction=USBDirection.OUT)
    @to_device
    def handle_another_request(self, request):

        #
        # Another set of convenience decorators exist to refine requests.
        # Decorators like `to_device` or `to_any_endpoint` chain with our
        # request decorators, and are syntax sugar for having an argument like
        # ``recipient=USBRequestRecipient.DEVICE`` in the handler decorator.
        #

        # For out requests, in lieu of a response, we typically want to acknowledge
        # the request. This can be accomplished by calling .acknowledge() or .ack()
        # on the request.
        request.ack()

        # Of course, if we want to let the host know we can't handle a request, we
        # may also choose to stall it. This is as simple as calling request.stall().


    #
    # Note that request handlers can be used on configurations, interfaces, and
    # endpoints as well. For the latter two cases, the decorators `to_this_interface`
    # and `to_this_endpoint` are convenient -- they tell a request to run only if
    # it's directed at that endpoint in particular, as selected by its ``index`` parameter.
    #


# FaceDancer ships with a default main() function that you can use to set up and run
# your device. It ships with some nice features -- including a ``--suggest`` function
# that can suggest pieces of boilerplate code that might be useful in device emulation.
#
```

(continues on next page)

(continued from previous page)

```
# main() will accept either the type of device to emulate, or an device instance.
# It'll also accept asyncio coroutines, in case you want to run things alongside the
# relevant device code. See e.g. `examples/rubber-ducky.py` for an example.
#
main(TemplateDevice)

```

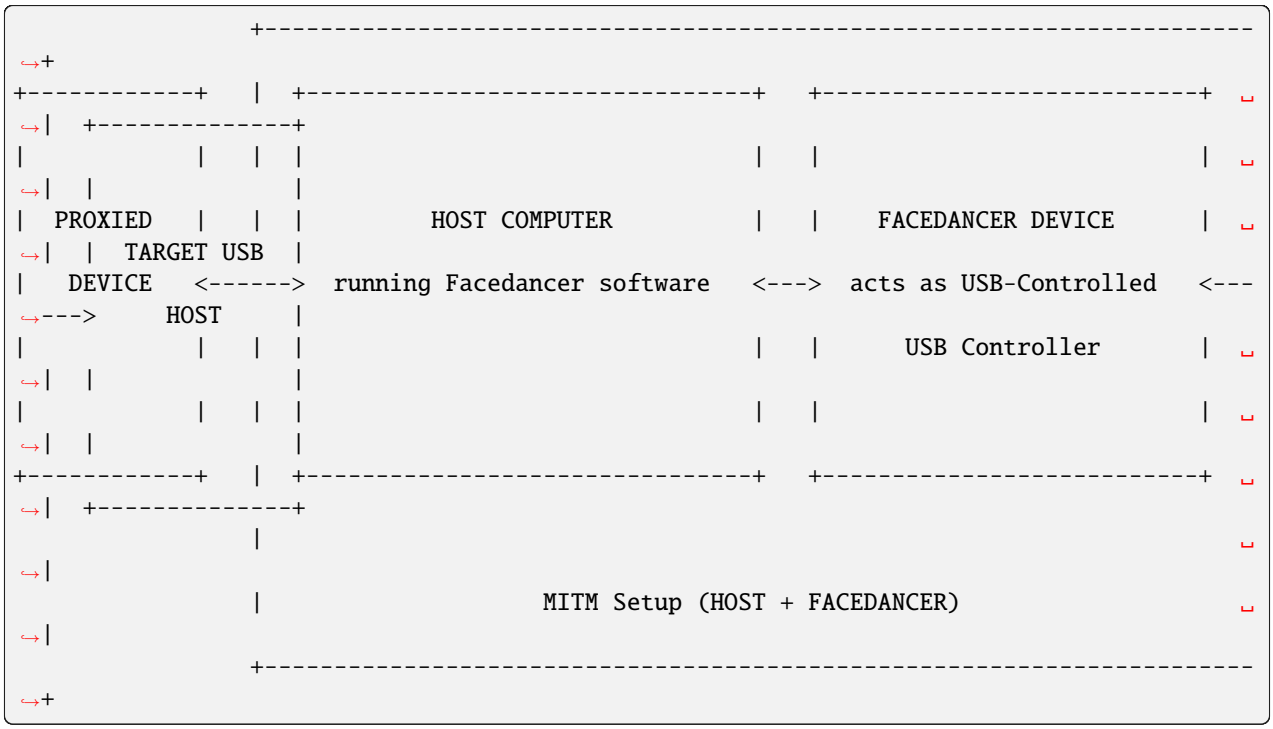
CHAPTER

FOUR

USING USB PROXY

4.1 Introduction

A major new feature of the newer Facedancer codebase is the ability to MITM (Meddler-In-The-Middle) USB connections – replacing the authors' original USBProxy project. This opens up a whole new realm of applications – including protocol analysis and live manipulation of USB packets – and is especially useful when you don't control the software running on the target device (e.g. on embedded systems or games consoles).



4.2 The Simplest USB Proxy

The simplest use for USB Proxy is to transparently forward USB transactions between the host to the device and log them to the console.

```
from facedancer           import *
from facedancer           import main

from facedancer.proxy     import USBProxyDevice
from facedancer.filters   import USBProxySetupFilters, USBProxyPrettyPrintFilter

# replace with the proxied device's information
ID_VENDOR=0x09e8
ID_PRODUCT=0x0031


if __name__ == "__main__":
    # create a USB Proxy Device
    proxy = USBProxyDevice(idVendor=ID_VENDOR, idProduct=ID_PRODUCT)

    # add a filter to forward control transfers between the target host and
    # proxied device
    proxy.add_filter(USBProxySetupFilters(proxy, verbose=0))

    # add a filter to log USB transactions to the console
    proxy.add_filter(USBProxyPrettyPrintFilter(verbose=5))

    main(proxy)
```

Setting up a USB Proxy begins by creating an instance of the *USBProxyDevice* with the vendor and product id's of the proxied device as arguments.

The actual behaviour of USB Proxy is governed by adding *filters* to the proxy that can intercept, read, modify and forward USB transactions between the host and device.

The first filter is a *USBProxySetupFilters* which is a simple forwarding filter that ensures all control transfers are forwarded between the target host and the proxied device. Without the presence of this script the target host will detect your proxied device but all attempts at enumeration would fail.

The second filter is a *USBProxyPrettyPrintFilter* which will intercept all transactions and then log them to the console.

4.3 Writing USB Proxy Filters

To write your own proxy filter you'd derive a new filter from *USBProxyFilter* and override the request handlers for the transactions you want to intercept.

For example, a simple filter to intercept and modify data from a MIDI controller could look like this:

```
from facedancer.filters import USBProxyFilter

class MyFilter(USBProxyFilter):
```

(continues on next page)

(continued from previous page)

```
    # intercept the midi controllers IN endpoint
    def filter_in(self, ep_num, data):

        # check if the data is from the correct endpoint and a midi message
        if ep_num == (0x82 & 0x7f) and len(data) == 4:

            # check if it is a midi note-on/off message
            if data[1] in [0x80, 0x90]:
                # transpose the note up by an octave - 7f
                data[2] += 12

        # return the endpoint number and modified data
        return ep_num, data
```

Which you can then add to the proxy using *USBProxyDevice*'s *add_filter()* method:

```
# add my filter to the proxy
proxy.add_filter(MyFilter())
```

You can find more information about the supported handlers in the *USBProxyFilter* documentation.

CHAPTER

FIVE

FACEDANCER EXAMPLES

There are a number of Facedancer examples available that demonstrate emulation of various USB device functions.

5.1 rubber-ducky.py

The canonical "Hello World" of USB emulation, the rubber-ducky example implements a minimal subset of the USB HID class specification in order to emulate a USB keyboard.

Table 1: Host Compatibility

Linux	macOS	Windows

5.2 ftdi-echo.py

An emulation of an FTDI USB-to-serial converter, the ftdi-echo example converts input received from a connected terminal to uppercase and echoes the result back to the sender.

Table 2: Host Compatibility

Linux	macOS	Windows

5.3 mass-storage.py

An emulation of a USB Mass Storage device, the mass-storage example can take a raw disk image file as input and present it to a target host as drive that can be mounted, read and written to.

You can create an empty disk image for use with the emulation using:

```
dd if=/dev/zero of=disk.img bs=1M count=100
mkfs -t ext4 disk.img
```

You can also test or modify the disk image locally by mounting it with:

```
mount -t auto -o loop disk.img /mnt
```

Remember to unmount it before using it with the device emulation!

Table 3: Host Compatibility

Linux	macOS	Windows

CHAPTER

SIX

FACEDANCER

6.1 facedancer package

6.1.1 Subpackages

facedancer.backends package

Submodules

facedancer.backends.MAXUSBApp module

class facedancer.backends.MAXUSBApp.**MAXUSBApp**(*device*, *verbose=0*)

Bases: *FacedancerApp*

static bytes_as_hex(*b*, *delim=' '*)

clear_irq_bit(*reg*, *bit*)

configured(*configuration*)

Callback that's issued when a USBDevice is configured, e.g. by the SET_CONFIGURATION request. Allows us to apply the new configuration.

Parameters

configuration – The configuration applied by the SET_CONFIG request.

connect(*usb_device*, *max_packet_size_ep0=64*, *device_speed=None*)

disconnect()

ep0_in_nak = 32

ep2_in_nak = 64

ep3_in_nak = 128

full_duplex = 16

get_version()

interrupt_level = 8

is_in0_buffer_avail = 1

is_in2_buffer_avail = 8

is_in3_buffer_avail = 16

is_out0_data_avail = 2

is_out1_data_avail = 4

is_setup_data_avail = 32

read_from_endpoint(*ep_num*)

reg_clr_togs = 10

reg_cpu_control = 16

reg_endpoint_interrupt_enable = 12

reg_endpoint_irq = 11

reg_ep0_byte_count = 5

reg_ep0_fifo = 0

reg_ep1_out_byte_count = 6

reg_ep1_out_fifo = 1

reg_ep2_in_byte_count = 7

reg_ep2_in_fifo = 2

reg_ep3_in_byte_count = 8

reg_ep3_in_fifo = 3

reg_ep_stalls = 9

reg_function_address = 19

reg_io_pins = 20

reg_pin_control = 17

reg_revision = 18

reg_setup_data_fifo = 4

reg_usb_control = 15

reg_usb_interrupt_enable = 14

reg_usb_irq = 13

send_on_endpoint(*ep_num*, *data*, *blocking=False*)

service_irqs()

set_address(*address*, *defer=False*)

Sets the device address of the Facedancer. Usually only used during initial configuration.

Parameters
address – The address that the Facedancer should assume.

stall_endpoint(*ep_number*, *direction=0*)

Stalls an arbitrary endpoint.

Parameters

- **ep_number** – The endpoint number to be stalled
- **direction** – 0 for out, 1 for in

stall_ep0(*direction=0*)

usb_control_connect = 8

usb_control_vbgate = 64

facedancer.backends.base module

class facedancer.backends.base.**FacedancerBackend**(*device: USBDevice = None*, *verbose: int = 0*, *quirks: List[str] = []*)

Bases: object

__init__(*device: USBDevice = None*, *verbose: int = 0*, *quirks: List[str] = []*)

Initializes the backend.

Parameters

- **device** – The device that will act as our Facedancer. (Optional)
- **verbose** – The verbosity level of the given application. (Optional)
- **quirks** – List of USB platform quirks. (Optional)

ack_status_stage(*direction: USBDirection = USBDirection.OUT*, *endpoint_number: int = 0*, *blocking: bool = False*)

Handles the status stage of a correctly completed control request, by priming the appropriate endpoint to handle the status phase.

Parameters

- **direction** – Determines if we're ACK'ing an IN or OUT vendor request. (This should match the direction of the DATA stage.)
- **endpoint_number** – The endpoint number on which the control request occurred.
- **blocking** – True if we should wait for the ACK to be fully issued before returning.

classmethod appropriate_for_environment(*backend_name: str*) → bool

Determines if the current environment seems appropriate for using this backend.

Parameters
backend_name – Backend name being requested. (Optional)

`configured`(*configuration: USBConfiguration*)

Callback that's issued when a USBDevice is configured, e.g. by the SET_CONFIGURATION request. Allows us to apply the new configuration.

Parameters
configuration – The USBConfiguration object applied by the SET_CONFIG request.

`connect`(*usb_device: USBDevice*, *max_packet_size_ep0: int = 64*, *device_speed: DeviceSpeed = DeviceSpeed.FULL*)

Prepares backend to connect to the target host and emulate a given device.

Parameters

- **usb_device** – The USBDevice object that represents the emulated device.
- **max_packet_size_ep0** – Max packet size for control endpoint.
- **device_speed** – Requested usb speed for the Facedancer board.

`disconnect`()

Disconnects Facedancer from the target host.

`get_version`()

Returns information about the active Facedancer version.

`read_from_endpoint`(*endpoint_number: int*) → bytes

Reads a block of data from the given endpoint.

Parameters
endpoint_number – The number of the OUT endpoint on which data is to be rx'd.

`reset`()

Triggers the Facedancer to handle its side of a bus reset.

`send_on_endpoint`(*endpoint_number: int*, *data: bytes*, *blocking: bool = True*)

Sends a collection of USB data on a given endpoint.

Parameters

- **endpoint_number** – The number of the IN endpoint on which data should be sent.
- **data** – The data to be sent.
- **blocking** – If true, this function should wait for the transfer to complete.

`service_irqs`()

Core routine of the Facedancer execution/event loop. Continuously monitors the Facedancer's execution status, and reacts as events occur.

`set_address`(*address: int*, *defer: bool = False*)

Sets the device address of the Facedancer. Usually only used during initial configuration.

Parameters

- **address** – The address the Facedancer should assume.
- **defer** – True iff the set_address request should wait for an active transaction to finish.

`stall_endpoint`(*endpoint_number: int*, *direction: USBDirection = USBDirection.OUT*)

Stalls the provided endpoint, as defined in the USB spec.

Parameters
endpoint_number – The number of the endpoint to be stalled.

facedancer.backends.goodfet module

class facedancer.backends.goodfet.**Facedancer**(*serialport*, *verbose=0*)

Bases: object

halt()

read(*n*)

Read raw bytes.

readcmd()

Read a single command.

reset()

write(*b*)

Write raw bytes.

writecmd(*c*)

Write a single command.

class facedancer.backends.goodfet.**FacedancerCommand**(*app=None*, *verb=None*, *data=None*)

Bases: object

as_bytestring()

long_string()

class facedancer.backends.goodfet.**GoodFETMonitorApp**(*device*, *verbose=0*)

Bases: *FacedancerApp*

announce_connected()

app_name = 'GoodFET monitor'

app_num = 0

echo(*s*)

get_clocking()

get_infostring()

list_apps()

print_info()

read_byte(*addr*)

facedancer.backends.goodfet.**GoodFETSerialPort**(***kwargs*)

Return a Serial port using default values possibly overriden by caller

class facedancer.backends.goodfet.**GoodfetMaxUSBApp**(*device=None*, *verbose=0*, *quirks=None*)

Bases: *MAXUSBApp*

ack_status_stage(*blocking=False*)

app_name = 'MAXUSB'

app_num = 64

classmethod appropriate_for_environment(*backend_name*)

Determines if the current environment seems appropriate for using the GoodFET::MaxUSB backend.

enable()

init_commands()

read_bytes(*reg*, *n*)

read_register(*reg_num*, *ack=False*)

write_bytes(*reg*, *data*)

write_register(*reg_num*, *value*, *ack=False*)

facedancer.backends.greatdancer module

class facedancer.backends.greatdancer.**GreatDancerApp**(*device=None*, *verbose=0*, *quirks=None*)

Bases: *FacedancerApp*

Backend for using GreatFET devices as FaceDancers.

DEVICE_TO_HOST = 1

GET_ENDPTCOMPLETE = 2

GET_ENDPTNAK = 4

GET_ENDPTSETUPSTAT = 1

GET_ENDPTSTATUS = 3

GET_USBSTS = 0

HOST_TO_DEVICE = 0

QUIRK_MANUAL_SET_ADDRESS = 1

SUPPORTED_ENDPOINTS = 4

USBSTS_D_NAKI = 65536

USBSTS_D_UI = 1

USBSTS_D_URI = 64

__init__(*device=None*, *verbose=0*, *quirks=None*)

Sets up a new GreatFET-backed Facedancer (GreatDancer) application.

device: The GreatFET device that will act as our GreatDancer. verbose: The verbosity level of the given application.

ack_status_stage(*direction=0*, *endpoint_number=0*, *blocking=False*)

Handles the status stage of a correctly completed control request, by priming the appropriate endpoint to handle the status phase.

Parameters

- **direction** – Determines if we're ACK'ing an IN or OUT vendor request. (This should match the direction of the DATA stage.)
- **endpoint_number** – The endpoint number on which the control request occurred.
- **blocking** – True if we should wait for the ACK to be fully issued before returning.

app_name = 'GreatDancer'

app_num = 0

classmethod appropriate_for_environment(*backend_name*)

Determines if the current environment seems appropriate for using the GreatDancer backend.

configured(*configuration*)

Callback that's issued when a USBDevice is configured, e.g. by the SET_CONFIGURATION request. Allows us to apply the new configuration.

Parameters
configuration – The configuration applied by the SET_CONFIG request.

connect(*usb_device*, *max_packet_size_ep0=64*, *device_speed=DeviceSpeed.FULL*)

Prepares the GreatDancer to connect to the target host and emulate a given device.

Parameters
usb_device – The USBDevice object that represents the device to be emulated.

disconnect()

Disconnects the GreatDancer from its target host.

get_version()

Returns information about the active GreatDancer version.

init_commands()

API compatibility function; not necessary for GreatDancer.

read_from_endpoint(*ep_num*)

Reads a block of data from the given endpoint.

Parameters
ep_num – The number of the OUT endpoint on which data is to be rx'd.

reset()

Triggers the GreatFET to handle its side of a bus reset.

send_on_endpoint(*ep_num*, *data*, *blocking=True*)

Sends a collection of USB data on a given endpoint.

Parameters

- **ep_num** – The number of the IN endpoint on which data should be sent.
- **data** – The data to be sent.
- **blocking** – If true, this function will wait for the transfer to complete.

service_irqs()

Core routine of the Facedancer execution/event loop. Continuously monitors the GreatDancer's execution status, and reacts as events occur.

set_address(*address*, *defer=False*)

Sets the device address of the GreatDancer. Usually only used during initial configuration.

Parameters

- **address** – The address that the GreatDancer should assume.
- **defer** – True iff the set_address request should wait for an active transaction to finish.

stall_endpoint(*ep_num*, *direction=0*)

Stalls the provided endpoint, as defined in the USB spec.

Parameters

ep_num – The number of the endpoint to be stalled.

stall_ep0(*direction=0*)

Convenience function that stalls the control endpoint zero.

facedancer.backends.greathost module

Host support for GreatFET-base devices.

class facedancer.backends.greathost.**GreatDancerHostApp**(*verbose=0*, *quirks=[]*, *autoconnect=True*, *device=None*)

Bases: *FacedancerUSBHost*

Class that represents a GreatFET-based USB host.

DEVICE_SPEED_FULL = 1

DEVICE_SPEED_HIGH = 2

DEVICE_SPEED_LOW = 0

DEVICE_SPEED_NAMES = {0: 'Low speed', 1: 'Full speed', 2: 'High speed', 3: 'Disconnected'}

DEVICE_SPEED_NONE = 3

DIRECTION_IN = 0

DIRECTION_OUT = 128

ENDPOINT_TYPE_CONTROL = 0

LINE_STATE_J = 1

LINE_STATE_K = 2

LINE_STATE_NAMES = {0: 'SE0', 1: 'J', 2: 'K', 3: 'No device / SE1'}

LINE_STATE_SE0 = 0

LINE_STATE_SE1 = 3

PID_IN = 1

PID_OUT = 0

PID_SETUP = 2

PORT_STATUS_REG = 0

PORT_STATUS_REGISTER_CONNECTED_MASK = 1

PORT_STATUS_REGISTER_ENABLED_MASK = 4

PORT_STATUS_REGISTER_LINE_STATE_MASK = 3

PORT_STATUS_REGISTER_LINE_STATE_SHIFT = 10

PORT_STATUS_REGISTER_POWERED_MASK = 4096

PORT_STATUS_REGISTER_SPEED_MASK = 3

PORT_STATUS_REGISTER_SPEED_SHIFT = 26

READ_STATUS_REG = 1

SPEED_REQUESTS = {0: 1, 1: 0, 2: 2, 3: 3}

STATUS_REG_SPEED_VALUES = {0: 1, 1: 0, 2: 2, 3: 3}

WRITE_STATUS_REG = 2

__init__(*verbose=0*, *quirks=[]*, *autoconnect=True*, *device=None*)

Sets up a GreatFET-based host connection.

app_name = 'GreatDancer Host'

classmethod appropriate_for_environment(*backend_name*)

Determines if the current environment seems appropriate for using the GreatDancer backend.

bus_reset(*delay=0.5*)

Issues a "bus reset", requesting that the downstream device reset itself.

Parameters

delay – The amount of time, in seconds, to wait before or after the reset request. To be compliant, this should be omitted, or set to 0.1s.

connect(*device_speed=None*)

Sets up our host to talk to the device, including turning on VBUS.

current_device_speed(*as_string=False*)

Returns the speed of the connected device

Parameters

as_string – If true, returns the speed as a string for printing; otherwise returns a DEVICE_SPEED_* constant.

current_line_state(*as_string=False*)

Returns the current state of the USB differential pair

Parameters

as_string – If true, returns the speed as a string for printing; otherwise returns a LINE_STATE_* constant.

device_is_connected()

Returns true iff a given device is connected.

initialize_control_endpoint(*device_address=None*, *device_speed=None*, *max_packet_size=None*)

Set up the device's control endpoint, so we can use it for e.g. enumeration.

port_is_enabled()

Returns true iff the FaceDancer host port's enabled.

port_is_powered()

Returns true iff the FaceDancer host port's enabled.

read_from_endpoint(*endpoint_number*, *expected_read_size=64*, *data_packet_pid=0*)

Sends a block of data on the provided endpoints.

Parameters

- **endpoint_number** – The endpoint number on which to send.
- **expected_read_size** – The expected amount of data to be read.
- **data_packet_pid** – The data packet PID to use (1 or 0). Ignored if the endpoint is set to automatically alternate data PIDs.

Raises an IOError on a communications error or stall.

send_on_endpoint(*endpoint_number*, *data*, *is_setup=False*, *blocking=True*, *data_packet_pid=0*)

Sends a block of data on the provided endpoints.

Parameters

- **endpoint_number** – The endpoint number on which to send.
- **data** – The data to be transmitted.
- **is_setup** – True iff this transfer should begin with a SETUP token.
- **blocking** – True iff this transaction should wait for the transaction to complete.
- **data_packet_pid** – The data packet PID to use (1 or 0). Ignored if the endpoint is set to automatically alternate data PIDs.

Raises an IOError on a communications error or stall.

set_up_endpoint(*endpoint_address_or_object*, *endpoint_type=None*, *max_packet_size=None*, *device_address=None*, *endpoint_speed=None*, *handle_data_toggle=None*, *is_control_endpoint=None*)

Sets up an endpoint for use. Can be used to initialize an endpoint or to update its parameters. Two forms exist:

Parameters
endpoint_object – a USBEndpoint object with the parameters to be populated

or

Parameters

- **endpoint_address** – the address of the endpoint to be setup; including the direction bit
- **endpoint_type** – one of the ENDPOINT_TYPE constants that specifies the transfer mode on the endpoint_address
- **max_packet_size** – the maximum packet size to be communicated on the given endpoint
- **device_address** – the address of the device to be communicated with; if not provided, the last address will be used.

- **endpoint_speed** – the speed of the packets to be communicated on the endpoint; should be a DEVICE_SPEED_* constant; if not provided, the last device's speed will be used.
- **handle_data_toggle** – true iff the hardware should automatically handle selection of data packet PIDs
- **is_control_endpoint** – true iff the given packet is a for a control endpoint

facedancer.backends.libusbhost module

Host support for accessing libusb with a Facedancer-like syntax.

class facedancer.backends.libusbhost.**LibUSBHostApp**(*verbose=0*, *quirks=[]*, *index=0*, ***kwargs*)

Bases: *FacedancerUSBHost*

Class that represents a libusb-based USB host.

__init__(*verbose=0*, *quirks=[]*, *index=0*, ***kwargs*)

Creates a new libusb backend for communicating with a target device.

app_name = 'LibUSB Host'

classmethod appropriate_for_environment(*backend_name*)

Determines if the current environment seems appropriate for using the libusb backend.

bus_reset(*delay=0*)

Issues a "bus reset", requesting that the downstream device reset itself.

Parameters

delay – The amount of time, in seconds, to wait before or after the reset request. To be compliant, this should be omitted, or set to 0.1s.

connect(*device_speed=None*)

Sets up our host to talk to the device, including turning on VBUS.

control_request_in(*request_type*, *recipient*, *request*, *value=0*, *index=0*, *length=0*)

Performs an IN control request.

Parameters

- **request_type** – Determines if this is a standard, class, or vendor request. Accepts a REQUEST_TYPE_* constant.
- **recipient** – Determines the context in which this command is interpreted. Accepts a REQUEST_RECIPIENT_* constant.
- **request** – The request number to be performed.
- **value, index** – The standard USB request arguments, to be included in the setup packet. Their meaning varies depending on the request.
- **length** – The maximum length of data expected in response, or 0 if we don't expect any data back.

control_request_out(*request_type*, *recipient*, *request*, *value=0*, *index=0*, *data=[]*)

Performs an OUT control request.

Parameters

- **request_type** – Determines if this is a standard, class, or vendor request. Accepts a REQUEST_TYPE_* constant.

- **recipient** – Determines the context in which this command is interpreted. Accepts a REQUEST_RECIPIENT_* constant.
- **request** – The request number to be performed.
- **value, index** – The standard USB request arguments, to be included in the setup packet. Their meaning varies depending on the request.
- **data** – The data to be transmitted with this control request.

`current_device_speed`(*as_string=False*)

Returns the speed of the connected device

Parameters

as_string – If true, returns the speed as a string for printing; otherwise returns a DEVICE_SPEED_* constant.

`current_line_state`(*as_string=False*)

Returns the current state of the USB differential pair

as_string

[If true, returns the speed as a string for printing; otherwise] returns a LINE_STATE_* constant.

`device_is_connected`()

Returns true iff a given device is connected.

`initialize_control_endpoint`(*device_address=None*, *device_speed=None*, *max_packet_size=None*)

Set up the device's control endpoint, so we can use it for e.g. enumeration.

`port_is_enabled`()

Returns true iff a given device is connected.

`port_is_powered`()

Returns true iff a given device is connected.

`read_from_endpoint`(*endpoint_number*, *expected_read_size=64*, *data_packet_pid=0*)

Sends a block of data on the provided endpoints.

Parameters

- **endpoint_number** – The endpoint number on which to send.
- **expected_read_size** – The expected amount of data to be read.
- **data_packet_pid** – The data packet PID to use (1 or 0). Ignored if the endpoint is set to automatically alternate data PIDs.

raises an IOError on a communications error or stall

`send_on_endpoint`(*endpoint_number*, *data*, *is_setup=False*, *blocking=True*, *data_packet_pid=0*)

Sends a block of data on the provided endpoints.

Parameters

- **endpoint_number** – The endpoint number on which to send.
- **data** – The data to be transmitted.
- **is_setup** – True iff this transfer should begin with a SETUP token.
- **blocking** – True iff this transaction should wait for the transaction to complete.
- **data_packet_pid** – The data packet PID to use (1 or 0). Ignored if the endpoint is set to automatically alternate data PIDs.

raises an IOError on a communications error or stall

set_up_endpoint(*endpoint_address_or_object*, *endpoint_type=None*, *max_packet_size=None*, *device_address=None*, *endpoint_speed=None*, *handle_data_toggle=None*, *is_control_endpoint=None*)

Sets up an endpoint for use. Can be used to initialize an endpoint or to update its parameters. Two forms exist:

Parameters
endpoint_object – a USBEndpoint object with the parameters to be populated

or

Parameters

- **endpoint_address** – the address of the endpoint to be setup; including the direction bit
- **endpoint_type** – one of the ENDPOINT_TYPE constants that specifies the transfer mode on the endpoint_address
- **max_packet_size** – the maximum packet size to be communicated on the given endpoint
- **device_address** – the address of the device to be communicated with; if not provided, the last address will be used
- **endpoint_speed** – the speed of the packets to be communicated on the endpoint; should be a DEVICE_SPEED_* constant; if not provided, the last device's speed will be used.
- **handle_data_toggle** – true iff the hardware should automatically handle selection of data packet PIDs
- **is_control_endpoint** – true iff the given packet is a for a control endpoint

facedancer.backends.moondancer module

class facedancer.backends.moondancer.**InterruptEvent**(*data: Tuple[int, int]*)

Bases: object

USB_BUS_RESET: int = 10

USB_RECEIVE_CONTROL: int = 11

USB_RECEIVE_PACKET: int = 12

USB_SEND_COMPLETE: int = 13

__init__(*data: Tuple[int, int]*)

Parses a tuple of two bytes representing an interrupt event into an InterruptEvent.

Parameters
data – A tuple of two bytes. The first byte is the interrupt code, the second is the endpoint number.

class facedancer.backends.moondancer.**MoondancerApp**(*device: USBDevice = None*, *verbose: int = 0*, *quirks: List[str] = []*)

Bases: *FacedancerApp*, *FacedancerBackend*

Backend for using Cynthion devices as FaceDancers.

SUPPORTED_ENDPOINTS = 16

__init__(*device: USBDevice = None*, *verbose: int = 0*, *quirks: List[str] = []*)

Sets up a new Cynthion-backed Facedancer (Moondancer) application.

Parameters

- **device** – The Cynthion device that will act as our Moondancer.
- **verbose** – The verbosity level of the given application.

ack_status_stage(*direction: USBDirection = USBDirection.OUT*, *endpoint_number: int = 0*, *blocking: bool = False*)

Handles the status stage of a correctly completed control request, by priming the appropriate endpoint to handle the status phase.

Parameters

- **direction** – Determines if we're ACK'ing an IN or OUT vendor request. (This should match the direction of the DATA stage.)
- **endpoint_number** – The endpoint number on which the control request occurred.
- **blocking** – True if we should wait for the ACK to be fully issued before returning.

app_name = 'Moondancer'

classmethod appropriate_for_environment(*backend_name: str*) → bool

Determines if the current environment seems appropriate for using the Moondancer backend.

configured(*configuration: USBConfiguration*)

Callback that's issued when a USBDevice is configured, e.g. by the SET_CONFIGURATION request. Allows us to apply the new configuration.

Parameters

configuration – The USBConfiguration object applied by the SET_CONFIG request.

connect(*usb_device: USBDevice*, *max_packet_size_ep0: int = 64*, *device_speed: DeviceSpeed = DeviceSpeed.FULL*)

Prepares Cynthion to connect to the target host and emulate a given device.

Parameters

usb_device – The USBDevice object that represents the device to be emulated.

disconnect()

Disconnects Cynthion from the target host.

get_version()

Returns information about the active Moondancer version.

handle_bus_reset()

Triggers Moondancer to perform its side of a bus reset.

handle_ep_in_nak_status(*nak_status: int*)

handle_receive_control(*endpoint_number: int*)

Handles a known outstanding control event on a given endpoint.

endpoint_number: The endpoint number for which a control event should be serviced.

handle_receive_packet(*endpoint_number: int*)

Handles a known-completed transfer on a given endpoint.

Parameters
endpoint_number – The endpoint number for which the transfer should be serviced.

`handle_send_complete`(*endpoint_number: int*)

`read_from_endpoint`(*endpoint_number: int*) → bytes

Reads a block of data from the given endpoint.

Parameters
endpoint_number – The number of the OUT endpoint on which data is to be rx'd.

`reset`()

Triggers the Cynthion to handle its side of a bus reset.

`send_on_endpoint`(*endpoint_number: int*, *data: bytes*, *blocking: bool = True*)

Sends a collection of USB data on a given endpoint.

Parameters

- **endpoint_number** – The number of the IN endpoint on which data should be sent.
- **data** – The data to be sent.
- **blocking** – If true, this function will wait for the transfer to complete.

`service_irqs`()

Core routine of the Facedancer execution/event loop. Continuously monitors the Moondancer's execution status, and reacts as events occur.

`set_address`(*address: int*, *defer: bool = False*)

Sets the device address of Moondancer. Usually only used during initial configuration.

Parameters

- **address** – The address that Moondancer should assume.
- **defer** – True iff the set_address request should wait for an active transaction to finish.

`stall_endpoint`(*endpoint_number: int*, *direction:* USBDirection *= USBDirection.OUT*)

Stalls the provided endpoint, as defined in the USB spec.

Parameters
endpoint_number – The number of the endpoint to be stalled.

`class facedancer.backends.moondancer.QuirkFlag`(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: `IntFlag`

`MANUAL_SET_ADDRESS: int = 1`

facedancer.backends.raspdancer module

class facedancer.backends.raspdancer.**Raspdancer**(*verbose=0*)

Bases: object

Extended version of the Facedancer class that accepts a direct SPI connection to the MAX324x chip, as used by the Raspdancer.

__init__(*verbose=0*)

Initializes our connection to the MAXUSB device.

reset()

Resets the connected MAXUSB chip.

set_up_comms()

Sets up the Raspdancer to communicate with the MAX324x.

transfer(*data*)

Emulate the facedancer's write command, which blasts data directly over to the SPI bus.

class facedancer.backends.raspdancer.**RaspdancerMaxUSBApp**(*device=None*, *verbose=0*, *quirks=None*)

Bases: *MAXUSBApp*

ack_status_stage(*blocking=False*)

app_name = 'MAXUSB'

app_num = 0

classmethod appropriate_for_environment(*backend_name*)

Determines if the current environment seems appropriate for using the GoodFET::MaxUSB backend.

enable()

init_commands()

read_bytes(*reg*, *n*)

read_register(*reg_num*, *ack=False*)

write_bytes(*reg*, *data*)

write_register(*reg_num*, *value*, *ack=False*)

Module contents

facedancer.classes package

Subpackages

facedancer.classes.hid package

Submodules

facedancer.classes.hid.descriptor module

Code for implementing HID classes.

facedancer.classes.hid.descriptor.**COLLECTION**(**octets*)

facedancer.classes.hid.descriptor.**DELIMITER**(**octets*)

facedancer.classes.hid.descriptor.**DESGINATOR_INDEX**(**octets*)

facedancer.classes.hid.descriptor.**DESGINATOR_MAXIMUM**(**octets*)

facedancer.classes.hid.descriptor.**DESGINATOR_MINIMUM**(**octets*)

facedancer.classes.hid.descriptor.**END_COLLECTION**()

facedancer.classes.hid.descriptor.**FEATURE**(*constant=False*, *variable=False*, *relative=False*, *wrap=False*, *nonlinear=False*, *preferred_state=True*, *nullable=False*, *buffered_bytes=False*)

class facedancer.classes.hid.descriptor.**HIDCollection**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

HID collections; from HID1.1 [6.2.2.4].

APPLICATION = 1

LOGICAL = 2

NAMED_ARRAY = 4

PHYSICAL = 0

REPORT = 3

USAGE_MODIFIER = 6

USAGE_SWITCH = 5

VENDOR = 255

class facedancer.classes.hid.descriptor.**HIDReportDescriptor**(*number: int = USBDescriptorTypeNumber.REPORT*, *raw: None = None*, *type_number: int = None*, *parent:* USBDescribable *= None*, *fields: Iterable[bytes] = ()*)

Bases: *USBDescriptor*

Descriptor class representing a HID report descriptor.

__call__(*index=0*)

Converts the descriptor object into raw bytes.

fields: Iterable[bytes] = ()

number: int = 34

raw: None = None

facedancer.classes.hid.descriptor.**INPUT**(*constant=False*, *variable=False*, *relative=False*, *wrap=False*, *nonlinear=False*, *preferred_state=True*, *nullable=False*, *buffered_bytes=False*)

facedancer.classes.hid.descriptor.**LOGICAL_MAXIMUM**(**octets*)

facedancer.classes.hid.descriptor.**LOGICAL_MINIMUM**(**octets*)

facedancer.classes.hid.descriptor.**OUTPUT**(*constant=False*, *variable=False*, *relative=False*, *wrap=False*, *nonlinear=False*, *preferred_state=True*, *nullable=False*, *buffered_bytes=False*)

facedancer.classes.hid.descriptor.**PHYSICAL_MAXIMUM**(**octets*)

facedancer.classes.hid.descriptor.**PHYSICAL_MINIMUM**(**octets*)

facedancer.classes.hid.descriptor.**POP**(**octets*)

facedancer.classes.hid.descriptor.**PUSH**(**octets*)

facedancer.classes.hid.descriptor.**REPORT_COUNT**(**octets*)

facedancer.classes.hid.descriptor.**REPORT_ID**(**octets*)

facedancer.classes.hid.descriptor.**REPORT_SIZE**(**octets*)

facedancer.classes.hid.descriptor.**STRING_INDEX**(**octets*)

facedancer.classes.hid.descriptor.**STRING_MAXIMUM**(**octets*)

facedancer.classes.hid.descriptor.**STRING_MINIMUM**(**octets*)

facedancer.classes.hid.descriptor.**UNIT**(**octets*)

facedancer.classes.hid.descriptor.**UNIT_EXPONENT**(**octets*)

facedancer.classes.hid.descriptor.**USAGE**(**octets*)

facedancer.classes.hid.descriptor.**USAGE_MAXIMUM**(**octets*)

facedancer.classes.hid.descriptor.**USAGE_MINIMUM**(**octets*)

facedancer.classes.hid.descriptor.**USAGE_PAGE**(**octets*)

facedancer.classes.hid.keyboard module

Helpers for HID keyboards.

class facedancer.classes.hid.keyboard.**KeyboardKeys**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

A = 4

AGAIN = 121

APOSTROPHE = 52

B = 5

BACKSLASH = 49

BACKSPACE = 42

C = 6

CAPSLOCK = 57

COMMA = 54

COMPOSE = 101

COPY = 124

CUT = 123

D = 7

DELETE = 76

DOT = 55

DOWN = 81

E = 8

END = 77

ENTER = 40

EQUAL = 46

ERR_OVF = 1

ESC = 41

F = 9

F1 = 58

F10 = 67

F11 = 68

F12 = 69

F13 = 104

F14 = 105

F15 = 106

F16 = 107

F17 = 108

F18 = 109

```
F19 = 110

F2 = 59

F20 = 111

F21 = 112

F22 = 113

F23 = 114

F24 = 115

F3 = 60

F4 = 61

F5 = 62

F6 = 63

F7 = 64

F8 = 65

F9 = 66

FIND = 126

FRONT = 119

G = 10

GRAVE = 53

H = 11

HANGEUL = 144

HANJA = 145

HASHTILDE = 50

HELP = 117

HENKAN = 138

HIRAGANA = 147

HOME = 74

I = 12

INSERT = 73

J = 13

K = 14

KATAKANA = 146
```

KATAKANAHIRAGANA = 136

KEYPAD_00 = 176

KEYPAD_000 = 177

KP0 = 98

KP1 = 89

KP2 = 90

KP3 = 91

KP4 = 92

KP5 = 93

KP6 = 94

KP7 = 95

KP8 = 96

KP9 = 97

KPASTERISK = 85

KPCOMMA = 133

KPDOT = 99

KPENTER = 88

KPEQUAL = 103

KPJPCOMMA = 140

KPLEFTPAREN = 182

KPMINUS = 86

KPPLUS = 87

KPRIGHTPAREN = 183

KPSLASH = 84

L = 15

LEFT = 80

LEFTALT = 226

LEFTBRACE = 47

LEFTCTRL = 224

LEFTMETA = 227

LEFTSHIFT = 225

```
M = 16

MEDIA_BACK = 241

MEDIA_CALC = 251

MEDIA_COFFEE = 249

MEDIA_EDIT = 247

MEDIA_EJECTCD = 236

MEDIA_FIND = 244

MEDIA_FORWARD = 242

MEDIA_MUTE = 239

MEDIA_NEXTSONG = 235

MEDIA_PLAYPAUSE = 232

MEDIA_PREVIOUSSONG = 234

MEDIA_REFRESH = 250

MEDIA_SCROLLDOWN = 246

MEDIA_SCROLLUP = 245

MEDIA_SLEEP = 248

MEDIA_STOP = 243

MEDIA_STOPCD = 233

MEDIA_VOLUMEDOWN = 238

MEDIA_VOLUMEUP = 237

MEDIA_WWW = 240

MINUS = 45

MUHENKAN = 139

MUTE = 127

N = 17

NONE = 0

NUMLOCK = 83

NUM_0 = 39

NUM_1 = 30

NUM_2 = 31

NUM_3 = 32
```

NUM_4 = 33

NUM_5 = 34

NUM_6 = 35

NUM_7 = 36

NUM_8 = 37

NUM_9 = 38

O = 18

OPEN = 116

P = 19

PAGEDOWN = 78

PAGEUP = 75

PASTE = 125

PAUSE = 72

POWER = 102

PROPS = 118

Q = 20

R = 21

RIGHT = 79

RIGHTALT = 230

RIGHTBRACE = 48

RIGHTCTRL = 228

RIGHTMETA = 231

RIGHTSHIFT = 229

RO = 135

S = 22

SCROLLLOCK = 71

SEMICOLON = 51

SLASH = 56

SPACE = 44

STOP = 120

SYSRQ = 70

T = 23

TAB = 43

U = 24

UNDO = 122

UP = 82

V = 25

VOLUMEDOWN = 129

VOLUMEUP = 128

W = 26

X = 27

Y = 28

YEN = 137

Z = 29

ZENKAKUHANKAKU = 148

classmethod get_scancode_for_ascii(*letter_or_code*)

Returns the (modifiers, scancode) used to type a given ASCII letter.

class facedancer.classes.hid.keyboard.**KeyboardModifiers**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntFlag

MOD_LEFT_ALT = 4

MOD_LEFT_CTRL = 1

MOD_LEFT_META = 8

MOD_LEFT_SHIFT = 2

MOD_RIGHT_ALT = 64

MOD_RIGHT_CTRL = 16

MOD_RIGHT_META = 128

MOD_RIGHT_SHIFT = 32

facedancer.classes.hid.usage module

Code for working with HID usages.

class facedancer.classes.hid.usage.**HIDGenericDesktopUsage**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

HID Usages for Generic Desktop Control; from [Table 6].

APPLICATION_BREAK = 165

APPLICATION_DEBUGGER_BREAK = 166

BYTE_COUNT = 59

COUNTED_BUFFER = 58

DIAL = 55

DPAD_DOWN = 145

DPAD_LEFT = 147

DPAD_RIGHT = 146

DPAD_UP = 144

FEATURE_NOTIFICATION = 71

GAMEPAD = 5

HAT_SWITCH = 57

JOYSTICK = 4

KEYBOARD = 6

KEYPAD = 7

MOTION_WAKEUP = 60

MOUSE = 2

MULTIAXIS_CONTROLLER = 8

POINTER = 1

RESOLUTION_MULTIPLIER = 72

RX = 51

RY = 52

RZ = 53

SELECT = 62

SLIDER = 54

```
START = 61

SYSTEM_APP_MENU = 134

SYSTEM_BREAK = 163

SYSTEM_COLD_RESTART = 142

SYSTEM_CONTEXT_MENU = 132

SYSTEM_CONTROL = 128

SYSTEM_DEBUGGER_BREAK = 164

SYSTEM_DISPLAY_AUTOSCALE = 183

SYSTEM_DISPLAY_BOTH = 179

SYSTEM_DISPLAY_DUAL = 180

SYSTEM_DISPLAY_EXTERNAL = 178

SYSTEM_DISPLAY_INTERNAL = 177

SYSTEM_DISPLAY_INVERT = 176

SYSTEM_DISPLAY_SWAP = 182

SYSTEM_DISPLAY_TOGGLE = 181

SYSTEM_DOCK = 160

SYSTEM_HIBERNATE = 168

SYSTEM_MAIN_MENU = 133

SYSTEM_MENU_DOWN = 141

SYSTEM_MENU_EXIT = 136

SYSTEM_MENU_HELP = 135

SYSTEM_MENU_LEFT = 139

SYSTEM_MENU_RIGHT = 138

SYSTEM_MENU_SELECT = 137

SYSTEM_MENU_UP = 140

SYSTEM_POWER_DOWN = 129

SYSTEM_SETUP = 162

SYSTEM_SLEEP = 130

SYSTEM_SPEAKER_MUTE = 167

SYSTEM_UNDOCK = 161

SYSTEM_WAKE_UP = 131
```

SYSTEM_WARM_UP = 143

TABLET_PC_SYSTEM_CONTROLS = 9

VBRX = 67

VBRY = 68

VBRZ = 69

VNO = 70

VX = 64

VY = 65

VZ = 66

WHEEL = 56

X = 48

Y = 49

Z = 50

class facedancer.classes.hid.usage.**HIDUsagePage**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

HID Usage Page numbers; from USB HID Usage Tables [Table 1].

ALPHANUMERIC_DISPLAY = 20

ARCADE = 145

BARCODE_SCANNER = 140

BUTTONS = 9

CAMERA_CONTROL = 144

CONSUMER = 12

DIGITIZER = 13

GAME = 5

GENERIC = 6

GENERIC_DESKTOP = 1

KEYBOARD = 7

LEDS = 8

MAGNETIC_STRIPE = 142

MEDICAL_INSTRUMENTS = 64

ORDINAL = 10

PID = 15

SCALE = 141

SIMULATION = 2

SPORT = 4

TELEPHONY = 11

UNICODE = 16

VENDOR_DEFINED = 65535

VR = 3

Module contents

Code for implementing HID classes.

Module contents

Support code for USB classes.

class facedancer.classes.**USBDeviceClass**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

Class representing known USB class numbers.

APPLICATION_SPECIFIC = 254

AUDIO = 1

AUDIO_VIDEO = 16

BILLBOARD = 17

CDC_DATA = 10

COMMUNICATIONS = 2

COMPOSITE = 0

CONTENT_SECURITY = 13

DIAGNOSTIC = 220

HID = 3

HUB = 9

IMAGE = 6

MASS_STORAGE = 8

MISCELLANEOUS = 239

PERSONAL_HEALTHCARE = 15

PHYSICAL = 5

PRINTER = 7

SMART_CARD = 11

TYPE_C_BRIDGE = 18

VENDOR_SPECIFIC = 255

VIDEO = 14

WIRELESS_CONTROLLER = 224

facedancer.devices package

Subpackages

facedancer.devices.umass package

Submodules

facedancer.devices.umass.disk_image module

class facedancer.devices.umass.disk_image.**DiskImage**

Bases: object

Class representing an arbitrary disk image, which can be procedurally generated, or which can be rendered from e.g. a file.

Currently limited to representing disk with 512-byte sectors.

close()

Closes and cleans up any resources held by the disk image.

get_data(*address*, *length*)

get_sector_count()

Returns the disk's sector count.

get_sector_data(*address*)

Returns the raw binary data for a given sector.

get_sector_size()

put_data(*address*, *data*)

put_sector_data(*address*, *data*)

Sets the raw binary data for a given disk sector.

class facedancer.devices.umass.disk_image.**FAT32DiskImage**(*size=268435456*, *verbose=0*)

Bases: *DiskImage*

Class for manufacturing synthetic FAT32 disk images.

BPB_SECTOR = 2048

CLUSTER_SIZE = 512

DATA_SECTION_START = 10146

FAT_END = 6113

FAT_START = 2080

FSINFO_SECTOR = 2049

MBR_SECTOR = 0

ROOT_DIR_ENTRY = 10146

get_partition_sectors()

Get the amount of sectors available for use by our main FAT partition.

get_sector_count()

Returns the total number of sectors present on the disk.

get_sector_data(*address*)

Fetches the data at the given sector of our emulated disk.

handle_bpb_read(*address*)

Returns a valid Boot Parameter Block, which tells the device how to interpret our FAT filesystem.

handle_fat_read(*address*)

Handles an access to the device's file allocation table.

handle_fsinfo_read(*address*)

Returns a valid filesystem info block, which is used to cache information about free sectors on the filesystem. We don't actually sport writing, so we return a valid-but-useless block.

handle_mbr_read(*address*)

Returns a master boot record directing the target device to our emulated FAT32 partition.

handle_root_dir_read(*address*)

Returns a valid entry describing the root directory of our FAT filesystem.

handle_unhandled_sector(*address*)

Handles unsupported sector reads.

class facedancer.devices.umass.disk_image.**RawDiskImage**(*filename*, *block_size*, *verbose=0*)

Bases: *DiskImage*

Raw disk image backed by a file.

close()

Closes and cleans up any resources held by the disk image.

get_sector_count()

Returns the disk's sector count.

get_sector_data(*address*)

Returns the raw binary data for a given sector.

put_data(*address*, *data*)

put_sector_data(*address*, *data*)

Sets the raw binary data for a given disk sector.

facedancer.devices.umass.umass module

Emulation of a USB Mass storage device.

class facedancer.devices.umass.umass.**CommandBlockWrapper**(*bytestring*)

Bases: object

class facedancer.devices.umass.umass.**ScsiCommandHandler**(*device*, *disk_image*, *verbose=0*)

Bases: object

STATUS_FAILURE = 2

STATUS_INCOMPLETE = -1

STATUS_OKAY = 0

continue_write(*cbw*, *data*)

handle_data_received(*data*)

handle_get_format_capacity(*cbw*)

handle_get_read_capacity(*cbw*)

handle_get_read_capacity_16(*cbw*)

handle_ignored_event(*cbw*)

Handles SCSI events that we can safely ignore.

handle_inquiry(*cbw*)

handle_mode_sense_10(*cbw*)

handle_mode_sense_6(*cbw*)

handle_read(*cbw*)

handle_read_16(*cbw*)

handle_scsi_command(*cbw*)

Handles an SCSI command.

handle_sense(*cbw*)

Handles SCSI sense requests.

handle_service_action_in(*cbw*)

handle_unknown_command(*cbw*)

Handles unsupported SCSI commands.

handle_write(*cbw*)

handle_write_16(*cbw*)

name: str = 'SCSI Command Handler'

class facedancer.devices.umass.umass.USBMassStorageDevice(*disk_image*)

Bases: *USBDevice*

Class implementing an emulated USB Mass Storage device.

connect()

Connects this device to the host; e.g. turning on our presence-detect pull up.

device_revision: int = 3

disconnect()

Disconnects this device from the host.

handle_bulk_only_mass_storage_reset_request = <ControlRequestHandler wrapping USBMassStorageDevice.handle_bulk_only_mass_storage_reset_request at 0x7fae21818470

handle_data_received(*endpoint*, *data*)

Handler for receipt of non-control request data.

Typically, this method will delegate any data received to the appropriate configuration/interface/endpoint. If overridden, the overriding function will receive all data.

Parameters

- **endpoint_number** – The endpoint number on which the data was received.
- **data** – The raw bytes received on the relevant endpoint.

handle_get_max_lun_request = <ControlRequestHandler wrapping USBMassStorageDevice.handle_get_max_lun_request at 0x7fae21818110

manufacturer_string: str = 'Facedancer'

max_packet_size_ep0: int = 64

name: str = 'USB mass storage interface'

product_id: int = 20561

product_string: str = 'USB Mass Storage emulation'

vendor_id: int = 33031

async wait_for_host()

Waits until the host connects by TODO.

facedancer.devices.umass.umass.bytes_as_hex(*b*, *delim=' '*)

Module contents

Submodules

facedancer.devices.ftdi module

Emulation of an FTDI USB-to-serial converter.

class facedancer.devices.ftdi.**FTDIDevice**(*name: str = 'generic device'*, *device_class: int = 0*, *device_subclass: int = 0*, *protocol_revision_number: int = 0*, *max_packet_size_ep0: int = 64*, *vendor_id: int = 1027*, *product_id: int = 24577*, *manufacturer_string: str = 'not-FTDI'*, *product_string: str = 'FTDI emulation'*, *serial_number_string: str = 'S/N 3420E'*, *supported_languages: tuple = (LanguageIDs.ENGLISH_US,)*, *device_revision: int = 1*, *usb_spec_version: int = 2*, *device_speed: ~facedancer.types.DeviceSpeed = None*, *descriptors: ~typing.Dict[int, bytes | callable] = <factory>*, *configurations: ~typing.Dict[int, ~facedancer.configuration.USBConfiguration] = <factory>*, *backend: ~facedancer.core.FacedancerUSBApp = None*)

Bases: *USBDevice*

Class implementing an emulated FTDI device.

device_revision: int = 1

handle_data_received(*endpoint*, *data*)

Called back whenever data is received.

handle_get_latency_timer_request = <ControlRequestHandler wrapping FTDIDevice.handle_get_latency_timer_request at 0x7fae21b84ce0

handle_get_modem_status_request = <ControlRequestHandler wrapping FTDIDevice.handle_get_modem_status_request at 0x7fae21b9aff0

handle_modem_ctrl_request = <ControlRequestHandler wrapping FTDIDevice.handle_modem_ctrl_request at 0x7fae21b9a390

handle_reset_request = <ControlRequestHandler wrapping FTDIDevice.handle_reset_request at 0x7fae21b9a7e0

handle_serial_data_received(*data*)

Callback executed when serial data is received.

Subclasses should override this to capture data from the host.

handle_set_baud_rate_request = <ControlRequestHandler wrapping FTDIDevice.handle_set_baud_rate_request at 0x7fae21b98e30

handle_set_data_request = <ControlRequestHandler wrapping FTDIDevice.handle_set_data_request at 0x7fae21b988c0

handle_set_error_char_request = <ControlRequestHandler wrapping FTDIDevice.handle_set_error_char_request at 0x7fae21b9bb30

handle_set_event_char_request = <ControlRequestHandler wrapping FTDIDevice.handle_set_event_char_request at 0x7fae21b9b3b0>

handle_set_flow_ctrl_request = <ControlRequestHandler wrapping FTDIDevice.handle_set_flow_ctrl_request at 0x7fae21b99a30>

handle_set_latency_timer_request = <ControlRequestHandler wrapping FTDIDevice.handle_set_latency_timer_request at 0x7fae21b9be30>

manufacturer_string: str = 'not-FTDI'

product_id: int = 24577

product_string: str = 'FTDI emulation'

reset_ftdi()

Resets the FTDI driver back to its original state.

transmit(*data: str | bytes*, ***, *blocking: bool = False*, *adjust_endings: bool = True*)

Transmits a block of data over the provided FTDI link to the host.

Parameters

- **data – The data to be sent.**
- **blocking – If true, this method will wait for completion before returning.**
- **adjust_endings – If true, line endings will be adjusted before sending.**

vendor_id: int = 1027

async wait_for_host()

Waits until the host connects by waiting for DTR assertion.

class facedancer.devices.ftdi.FTDIFlowControl(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntFlag

Constants describing how FTDI flow control works.

DTR_DSR = 2

NO_FLOW_CONTROL = 0

RTS_CTS = 1

XON_XOFF = 4

facedancer.devices.keyboard module

class facedancer.devices.keyboard.**USBKeyboardDevice**(*name: str = 'USB keyboard device'*, *device_class: int = 0*, *device_subclass: int = 0*, *protocol_revision_number: int = 0*, *max_packet_size_ep0: int = 64*, *vendor_id: int = 24843*, *product_id: int = 18003*, *manufacturer_string: str = 'FaceDancer'*, *product_string: str = 'Non-suspicious Keyboard'*, *serial_number_string: str = 'S/N 3420E'*, *supported_languages: tuple = (LanguageIDs.ENGLISH_US,)*, *device_revision: int = 0*, *usb_spec_version: int = 2*, *device_speed: ~facedancer.types.DeviceSpeed = None*, *descriptors: ~typing.Dict[int, bytes | callable] = <factory>*, *configurations: ~typing.Dict[int, ~facedancer.configuration.USBConfiguration] = <factory>*, *backend: ~facedancer.core.FacedancerUSBApp = None*)

Bases: *USBDevice*

Simple USB keyboard device.

KeyboardConfiguration = <facedancer.magic.AutoInstantiator object>

all_keys_up(*, *include_modifiers: bool = True*)

Releases all keys currently pressed.

Parameters
include_modifiers – If set to false, modifiers will be left at their current states.

all_modifiers_up()

Releases all modifiers currently held.

handle_data_requested(*endpoint:* USBEndpoint)

Provide data once per host request.

key_down(*code:* KeyboardKeys)

Marks a given key as pressed; should be a scancode from KeyboardKeys.

key_up(*code:* KeyboardKeys)

Marks a given key as released; should be a scancode from KeyboardKeys.

modifier_down(*code:* KeyboardModifiers)

Marks a given modifier as pressed; should be a flag from KeyboardModifiers.

modifier_up(*code:* KeyboardModifiers)

Marks a given modifier as released; should be a flag from KeyboardModifiers.

name: str = 'USB keyboard device'

product_string: str = 'Non-suspicious Keyboard'

async type_letter(*letter: str*, *duration: float = 0.1*, *modifiers:* KeyboardModifiers *= None*)

Attempts to type a single letter, based on its ASCII string representation.

Parameters

- **letter – A single-character string literal, to be typed.**

- **duration – How long each key should be pressed, in seconds.**
- **modifiers – Any modifier keys that should be held while typing.**

async type_letters(**letters: Iterable[str]*, *duration: float = 0.1*)

Attempts to type a string of letters, based on ASCII string representations.

Parameters

- ***letters – A collection of single-character string literal, to be typed in order.**
- **duration – How long each key should be pressed, in seconds.**

async type_scancode(*code:* KeyboardKeys, *duration: float = 0.1*, *modifiers:* KeyboardModifiers = *None*)

Presses, and then releases, a single key.

Parameters

- **code – The keyboard key to be pressed's scancode.**
- **duration – How long the given key should be pressed, in seconds.**
- **modifiers – Any modifier keys that should be held while typing.**

async type_scancodes(**codes: Iterable[*KeyboardKeys*]*, *duration: float = 0.1*)

Presses, and then releases, a collection of keys, in order.

Parameters

- ***code – The keyboard keys to be pressed's scancodes.**
- **duration – How long each key should be pressed, in seconds.**

async type_string(*to_type: str*, ***, *duration: float = 0.1*, *modifiers:* KeyboardModifiers = *None*)

Attempts to type a python string into the remote host.

Parameters

- **letter – A collection of single-character string literal, to be typed in order.**
- **duration – How long each key should be pressed, in seconds.**
- **modifiers – Any modifier keys that should be held while typing.**

Module contents

facedancer.devices.**default_main**(*device_or_type*, **coroutines*)

Simple, default main for FaceDancer emulation.

Parameters

device_type – The USBDevice type to emulate.

facedancer.filters package

Submodules

facedancer.filters.base module

class facedancer.filters.base.**USBProxyFilter**

Bases: object

Base class for filters that modify USB data.

filter_control_in(*request*, *data*, *stalled*)

Filters the data response from the proxied device during an IN control request. This allows us to modify the data returned from the proxied device during a setup stage.

Parameters

- **request** – The request that was issued to the target host.
- **data** – The data being proxied during the data stage.
- **stalled** – True if the proxied device (or a previous filter) stalled the request.

Returns

Modified versions of the arguments. Note that modifying request will _only_ modify the request as seen by future filters, as the SETUP stage has already passed and the request has already been sent to the device.

filter_control_in_setup(*request*, *stalled*)

Filters a SETUP stage for an IN control request. This allows us to modify the SETUP stage before it's proxied to the real device.

Parameters

- **request** – The request to be issued.
- **stalled** – True iff the packet has been stalled by a previous filter.

Returns

Modified versions of the arguments. If stalled is set to true, the packet will be immediately stalled an not proxied. If stalled is false, but request is returned as None, the packet will be NAK'd instead of proxied.

filter_control_out(*request*, *data*)

Filters handling of an OUT control request, which contains both a request and (optional) data stage.

Parameters

- **request** – The request issued by the target host.
- **data** – The data sent by the target host with the request.

Returns

Modified versions of the arguments. Returning a request of None will absorb the packet silently and not proxy it to the device.

filter_in(*ep_num*, *data*)

Filters the response to an IN token (the data packet received in response to the host issuing an IN token).

Parameters

- **ep_num** – The endpoint number associated with the data packet.

- **data** – The data packet received from the proxied device.

Returns
A modified version of the arguments. If data is set to none, the packet will be absorbed, and a NAK will be issued instead of responding to the IN request with data.

`filter_in_token`(*ep_num*)

Filters an IN token before it's passed to the proxied device. This allows modification of e.g. the endpoint or absorption of the IN token before it's issued to the real device.

Parameters
ep_num – The endpoint number on which the IN token is to be proxied.

Returns
A modified version of the arguments. If ep_num is set to None, the token will be absorbed and not issued to the target host.

`filter_out`(*ep_num*, *data*)

Filters a packet sent from the host via an OUT token.

Parameters

- **ep_num** – The endpoint number associated with the data packet.
- **data** – The data packet received from host.

Returns
A modified version of the arguments. If data is set to none, the packet will be absorbed,

`handle_out_request_stall`(*request*, *data*, *stalled*)

Handles an OUT request that was stalled by the proxied device.

Parameters

- **request** – The request header for the request that stalled.
- **data** – The data stage for the request that stalled, if appropriate.
- **stalled** – True iff the request is still considered stalled. This can be overridden by previous filters, so it's possible for this to be false.

`handle_out_stall`(*ep_num*, *data*, *stalled*)

Handles an OUT transfer that was stalled by the victim.

Parameters

- **ep_num** – The endpoint number for the data that stalled.
- **data** – The data for the transfer that stalled, if appropriate.
- **stalled** – True iff the transfer is still considered stalled. This can be overridden by previous filters, so it's possible for this to be false.

facedancer.filters.logging module

class facedancer.filters.logging.**USBProxyPrettyPrintFilter**(*verbose=4*, *decoration=''*)

Bases: *USBProxyFilter*

Filter that pretty prints USB transactions according to log levels.

__init__(*verbose=4*, *decoration=''*)

Sets up a new USBProxy pretty printing filter.

filter_control_in(*req*, *data*, *stalled*)

Log IN control requests without modification.

filter_control_out(*req*, *data*)

Log OUT control requests without modification.

filter_in(*ep_num*, *data*)

Log IN transfers without modification.

filter_out(*ep_num*, *data*)

Log OUT transfers without modification.

handle_out_request_stall(*req*, *data*, *stalled*)

Handles cases where OUT requests are stalled (and thus we don't get data).

timestamp()

Generate a quick timestamp for printing.

facedancer.filters.standard module

Standard filters for USBProxy that should (almost) always be used.

class facedancer.filters.standard.**USBProxySetupFilters**(*device*, *verbose=0*)

Bases: *USBProxyFilter*

DESCRIPTOR_CONFIGURATION = 2

DESCRIPTOR_DEVICE = 1

GET_DESCRIPTOR_REQUEST = 6

MAX_PACKET_SIZE_EP0 = 64

RECIPIENT_DEVICE = 0

SET_ADDRESS_REQUEST = 5

SET_CONFIGURATION_REQUEST = 9

filter_control_in(*req*, *data*, *stalled*)

Filters the data response from the proxied device during an IN control request. This allows us to modify the data returned from the proxied device during a setup stage.

Parameters

- **request** – The request that was issued to the target host.
- **data** – The data being proxied during the data stage.

- **stalled** – True if the proxied device (or a previous filter) stalled the request.

Returns
Modified versions of the arguments. Note that modifying request will _only_ modify the request as seen by future filters, as the SETUP stage has already passed and the request has already been sent to the device.

`filter_control_out`(*req*, *data*)

Filters handling of an OUT control request, which contains both a request and (optional) data stage.

Parameters

- **request** – The request issued by the target host.
- **data** – The data sent by the target host with the request.

Returns
Modified versions of the arguments. Returning a request of None will absorb the packet silently and not proxy it to the device.

Module contents

6.1.2 Submodules

6.1.3 facedancer.configuration module

Functionality for describing USB device configurations.

`class facedancer.configuration.USBConfiguration`(*number: int = 1*, *configuration_string: str = None*, *max_power: int = 500*, *self_powered: bool = True*, *supports_remote_wakeup: bool = True*, *parent: ~facedancer.descriptor.USBDescribable = None*, *interfaces: ~facedancer.interface.USBInterface = <factory>*)

Bases: *USBDescribable*, *AutoInstantiable*, *USBRequestHandler*

Class representing a USBDevice's configuration.

Fields:

number:
The configuration's number; one-indexed.

configuration_string
A string describing the configuration; or None if not provided.

max_power:
The maximum power expected to be drawn by the device when using this interface, in mA. Typically 500mA, for maximum possible.

supports_remote_wakeup:
True iff this device should be able to wake the host from suspend.

`DESCRIPTOR_SIZE_BYTES = 9`

`DESCRIPTOR_TYPE_NUMBER = 2`

`add_interface`(*interface:* USBInterface)

Adds an interface to the configuration.

property attributes

Retrives the "attributes" composite word.

configuration_string: str = None

classmethod from_binary_descriptor(*data*)

Generates a new USBConfiguration object from a configuration descriptor, handling any attached subordinate descriptors.

Parameters
data – The raw bytes for the descriptor to be parsed.

get_descriptor() → bytes

Returns this configuration's configuration descriptor, including subordinates.

get_device()

Returns a reference to the associated device.

get_endpoint(*number: int*, *direction:* USBDirection) → *USBEndpoint*

Attempts to find an endpoint with the given number + direction.

Parameters

- **number** – The endpoint number to look for.
- **direction** – Whether to look for an IN or OUT endpoint.

get_identifier() → int

Returns a unique integer identifier for this object.

This is usually the index or address of the relevant USB object.

get_interfaces() → Iterable[*USBInterface*]

Returns an iterable over all interfaces on the provided device.

handle_buffer_empty(*endpoint:* USBEndpoint)

Handler called when a given endpoint first has an empty buffer.

Often, an empty buffer indicates an opportunity to queue data for sending ('prime an endpoint'), but doesn't necessarily mean that the host is planning on reading the data.

This function is called only once per buffer.

handle_data_received(*endpoint:* USBEndpoint, *data: bytes*)

Handler for receipt of non-control request data.

Typically, this method will delegate any data received to the appropriate configuration/interface/endpoint. If overridden, the overriding function will receive all data; and can delegate it by calling the *.handle_data_received* method on *self.configuration*.

Parameters

- **endpoint** – The endpoint on which the data was received.
- **data** – The raw bytes received on the relevant endpoint.

handle_data_requested(*endpoint:* USBEndpoint)

Handler called when the host requests data on a non-control endpoint.

Typically, this method will delegate the request to the appropriate interface+endpoint. If overridden, the overriding function will receive all data.

Parameters
endpoint – The endpoint on which the host requested data.

interfaces: *USBInterface*

max_power: int = 500

number: int = 1

parent: *USBDescribable* = None

self_powered: bool = True

supports_remote_wakeup: bool = True

6.1.4 facedancer.core module

class facedancer.core.**FacedancerApp**(*device*, *verbose=0*)

Bases: object

app_name = 'override this'

app_num = 0

classmethod appropriate_for_environment(*backend_name=None*)

Returns true if the current class is likely to be the appropriate class to connect to a facedancer given the board_name and other environmental factors.

Parameters
backend_name – The name of the backend, as typically retrieved from the BACKEND environment variable, or None to try figuring things out based on other environmental factors.

classmethod autodetect(*verbose=0*, *quirks=None*)

Convenience function that automatically creates the appropriate subclass based on the BOARD environment variable and some crude internal automagic.

Parameters
verbose – Sets the verbosity level of the relevant app. Increasing this from zero yields progressively more output.

enable()

init_commands()

class facedancer.core.**FacedancerBasicScheduler**

Bases: object

Most basic scheduler for Facedancer devices– and the schedule which is created implicitly if no other scheduler is provided. Executes each of its tasks in order, over and over.

add_task(*callback*)

Adds a facedancer task to the scheduler, which will be called repeatedly according to the internal scheduling algorithm

callback: The callback to be scheduled.

do_exit = False

run()

Run the main scheduler stack.

stop()

Stop the scheduler on next loop.

facedancer.core.**FacedancerUSBApp**(*verbose=0*, *quirks=None*)

Convenience function that automatically creates a FacedancerApp based on the BOARD environment variable and some crude internal automagic.

Parameters

verbose – Sets the verbosity level of the relevant app. Increasing this from zero yields progressively more output.

class facedancer.core.**FacedancerUSBHost**

Bases: object

Base class for FaceDancer host connections– extended to provide actual connections to each host.

ENDPOINT_DIRECTION_IN = 128

ENDPOINT_DIRECTION_OUT = 0

ENDPOINT_TYPE_CONTROL = 0

PID_IN = 1

PID_OUT = 0

PID_SETUP = 2

REQUEST_RECIPIENT_DEVICE = 0

REQUEST_RECIPIENT_ENDPOINT = 2

REQUEST_RECIPIENT_INTERFACE = 1

REQUEST_RECIPIENT_OTHER = 3

REQUEST_TYPE_CLASS = 1

REQUEST_TYPE_RESERVED = 3

REQUEST_TYPE_STANDARD = 0

REQUEST_TYPE_VENDOR = 2

STANDARD_REQUEST_GET_DESCRIPTOR = 6

STANDARD_REQUEST_GET_STATUS = 0

STANDARD_REQUEST_SET_ADDRESS = 5

STANDARD_REQUEST_SET_CONFIGURATION = 9

apply_configuration(*index*, *set_configuration=True*)

Applies a device's configuration. Necessary to use endpoints other than the control endpoint.

Parameters

- **index** – The configuration index to apply.
- **set_configuration** – If true, also informs the device of the change. Setting this to false can allow the host to update its view of all endpoints without communicating with the device – e.g. to update the device's address.

`classmethod appropriate_for_environment`(*backend_name=None*)

Returns true if the current class is likely to be the appropriate class to connect to a facedancer given the board_name and other environmental factors.

Parameters

backend_name – The name of the backend, as typically retrieved from the BACKEND environment variable, or None to try figuring things out based on other environmental factors.

`classmethod autodetect`(*verbose=0*, *quirks=None*)

Convenience function that automatically creates the appropriate subclass based on the BOARD environment variable and some crude internal automagic.

Parameters

verbose – Sets the verbosity level of the relevant app. Increasing this from zero yields progressively more output.

`control_request_in`(*request_type*, *recipient*, *request*, *value=0*, *index=0*, *length=0*)

Performs an IN control request.

Parameters

- **request_type** – Determines if this is a standard, class, or vendor request. Accepts a REQUEST_TYPE_* constant.
- **recipient** – Determines the context in which this command is interpreted. Accepts a REQUEST_RECIPIENT_* constant.
- **request** – The request number to be performed.
- **value, index** – The standard USB request arguments, to be included in the setup packet. Their meaning varies depending on the request.
- **length** – The maximum length of data expected in response, or 0 if we don't expect any data back.

`control_request_out`(*request_type*, *recipient*, *request*, *value=0*, *index=0*, *data=[]*)

Performs an OUT control request.

Parameters

- **request_type** – Determines if this is a standard, class, or vendor request. Accepts a REQUEST_TYPE_* constant.
- **recipient** – Determines the context in which this command is interpreted. Accepts a REQUEST_RECIPIENT_* constant.
- **request** – The request number to be performed.
- **value, index** – The standard USB request arguments, to be included in the setup packet. Their meaning varies depending on the request.
- **data** – The data to be transmitted with this control request.

`get_configuration_descriptor`(*index=0*, *include_subordinates=True*)

Returns the device's configuration descriptor.

Parameters
include_subordinate – if true, subordinate descriptors will also be returned

`get_descriptor`(*descriptor_type*, *descriptor_index*, *language_id*, *max_length*)

Reads up to max_length bytes of a device's descriptors.

`get_device_descriptor`(*max_length=18*)

Returns the device's device descriptor.

`handle_events`()

`initialize_device`(*apply_configuration=0*, *assign_address=0*)

Sets up a connection to a directly-attached USB device.

Parameters

- **apply_configuration** – If non-zero, the configuration with the given index will be applied to the relevant device.
- **assign_address** – If non-zero, the device will be assigned the given address as part of the enumeration/initialization process.

`read_ep0_max_packet_size`()

Returns the device's reported maximum packet size on EP0, in a way appropriate for an barely-configured endpoint.

`set_address`(*device_address*)

Sets the device's address.

Note that all endpoints must be set up again after issuing the new address; the easiest way to do this is to call apply_configuration().

Parameters
device_address – the address to apply to the given device

`set_configuration`(*index*)

Sets the device's active configuration.

Note that this does not configure the host for the given configuration. Most of the time, you probably want apply_configuration, which does.

Parameters
index – the index of the configuration to apply

facedancer.core.`FacedancerUSBHostApp`(*verbose=0*, *quirks=None*)

Convenience function that automatically creates a FacedancerApp based on the BOARD environment variable and some crude internal automagic.

verbose: Sets the verbosity level of the relevant app. Increasing
this from zero yields progressively more output.

6.1.5 facedancer.descriptor module

Functionality for working with objects with associated USB descriptors.

class facedancer.descriptor.**StringDescriptorManager**

Bases: object

Manager class that collects active string descriptors.

__getitem__(*index*)

Gets the relevant string descriptor.

add_string(*string*)

Adds a python string to the string manager, and returns an index.

get_index(*string*)

Returns the index of the given string; creating it if the string isn't already known.

class facedancer.descriptor.**USBClassDescriptor**(*number: int*, *raw: bytes*, *type_number: int = None*, *parent: USBDescribable = None*)

Bases: *USBDescriptor*

Class for arbitrary USB Class descriptors.

class facedancer.descriptor.**USBDescribable**

Bases: object

Abstract base class for objects that can be created from USB descriptors.

DESCRIPTOR_TYPE_NUMBER = None

classmethod from_binary_descriptor(*data*)

Attempts to create a USBDescriptor subclass from the given raw descriptor data.

classmethod handles_binary_descriptor(*data*)

Returns true iff this class handles the given descriptor. By default, this is based on the class's DESCRIPTOR_TYPE_NUMBER declaration.

class facedancer.descriptor.**USBDescriptor**(*number: int*, *raw: bytes*, *type_number: int = None*, *parent: USBDescribable = None*)

Bases: *USBDescribable*, *AutoInstantiable*

Class for arbitrary USB descriptors; minimal concrete implementation of USBDescribable.

__call__(*index=0*)

Converts the descriptor object into raw bytes.

get_identifier()

Returns a unique integer identifier for this object.

This is usually the index or address of the relevant USB object.

number: int

parent: *USBDescribable* = None

raw: bytes

type_number: int = None

class facedancer.descriptor.**USBDescriptorTypeNumber**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

CONFIGURATION = 2

DEVICE = 1

DEVICE_QUALIFIER = 6

ENDPOINT = 5

HID = 33

INTERFACE = 4

INTERFACE_POWER = 8

OTHER_SPEED_CONFIGURATION = 7

REPORT = 34

STRING = 3

class facedancer.descriptor.**USBStringDescriptor**(*number: int*, *raw: bytes*, *type_number: int = None*, *parent: USBDescribable = None*, *python_string: str = None*)

Bases: *USBDescriptor*

Class representing a USB string descriptor.

DESCRIPTOR_TYPE_NUMBER = 3

classmethod from_string(*string*, ***, *index=None*)

python_string: str = None

6.1.6 facedancer.device module

Functionality for defining USB devices.

class facedancer.device.**USBBaseDevice**(*name: str = 'generic device'*, *device_class: int = 0*, *device_subclass: int = 0*, *protocol_revision_number: int = 0*, *max_packet_size_ep0: int = 64*, *vendor_id: int = 24843*, *product_id: int = 18003*, *manufacturer_string: str = 'FaceDancer'*, *product_string: str = 'Generic USB Device'*, *serial_number_string: str = 'S/N 3420E'*, *supported_languages: tuple = (LanguageIDs.ENGLISH_US,)*, *device_revision: int = 0*, *usb_spec_version: int = 2*, *device_speed: ~facedancer.types.DeviceSpeed = None*, *descriptors: ~typing.Dict[int, bytes | callable] = <factory>*, *configurations: ~typing.Dict[int, ~facedancer.configuration.USBConfiguration] = <factory>*, *backend: ~facedancer.core.FacedancerUSBApp = None*)

Bases: *USBDescribable*, *USBRequestHandler*

Base-most class for FaceDancer USB devices. This version is very similar to the USBDevice type, except that it does not define _any_ standard handlers. This allows you the freedom to declare whatever standard requests you'd like.

Fields:

vendor_id, product_id :
The USB vendor and product ID for this device.

manufacturer_string, product_string, serial_number_string :
Python strings identifying the device to the USB host.

device_class, device_subclass, protocol_revision_number :
The USB descriptor fields that select the class, subclass, and protocol.

supported_languages :
A tuple containing all of the language IDs supported by the device.

device_revision :
Number indicating the hardware revision of this device. Typically BCD.

usb_spec_revision :
Number indicating the version of the USB specification we adhere to. Typically 0x0200.

device_speed :
Specify the device speed for boards that support multiple interface speeds.

DESCRIPTOR_LENGTH = 18

DESCRIPTOR_TYPE_NUMBER = 1

__post_init__()
Set up our device for execution.

add_configuration(*configuration:* USBConfiguration)
Adds the provided configuration to this device.

backend: FacedancerUSBApp = None

configurations: Dict[int, *USBConfiguration*]

connect(*device_speed:* DeviceSpeed = *DeviceSpeed.FULL*)
Connects this device to the host; e.g. turning on our presence-detect pull up.

create_request(*raw_data: bytes*) → *USBControlRequest*

descriptors: Dict[int, bytes | callable]

device_class: int = 0

device_revision: int = 0

device_speed: *DeviceSpeed* = None

device_subclass: int = 0

disconnect()
Disconnects this device from the host.

emulate(**coroutines: Iterable[Coroutine]*)

Convenience method that runs a full method in a blocking manner. Performs connect, run, and then disconnect.

Parameters
***coroutines** – any asyncio coroutines to be executed concurrently with our emulation

classmethod from_binary_descriptor(*data*)

Creates a USBBaseDevice object from its descriptor.

get_configuration_descriptor(*index: int*) → bytes

Returns the configuration descriptor with the given configuration number.

get_descriptor() → bytes

Returns a complete descriptor for this device.

get_endpoint(*endpoint_number: int*, *direction:* USBDirection) → *USBEndpoint*

Attempts to find a subordinate endpoint matching the given number/direction.

Parameters

- **endpoint_number** – The endpoint number to search for.
- **direction** – The endpoint direction to be matched.

Returns
The matching endpoint; or None if no matching endpoint existed.

get_string_descriptor(*index: int*) → bytes

Returns the string descriptor associated with a given index.

handle_buffer_available(*ep_num*)

Backend data-buffer-empty handler; for legacy compatibility.

Prefer overriding handle_buffer_available().

handle_buffer_empty(*endpoint:* USBEndpoint)

Handler called when a given endpoint first has an empty buffer.

Often, an empty buffer indicates an opportunity to queue data for sending ('prime an endpoint'), but doesn't necessarily mean that the host is planning on reading the data.

This function is called only once per buffer.

handle_bus_reset()

Event handler for a bus reset.

handle_data_available(*ep_num*, *data*)

Backend data-available handler; for legacy compatibility.

Prefer overriding handle_data_received().

handle_data_received(*endpoint:* USBEndpoint, *data: bytes*)

Handler for receipt of non-control request data.

Typically, this method will delegate any data received to the appropriate configuration/interface/endpoint. If overridden, the overriding function will receive all data.

Parameters

- **endpoint_number** – The endpoint number on which the data was received.
- **data** – The raw bytes received on the relevant endpoint.

handle_data_requested(*endpoint:* USBEndpoint)

Handler called when the host requests data on a non-control endpoint.

Typically, this method will delegate the request to the appropriate configuration+interface+endpoint. If overridden, the overriding function will receive all events.

Parameters
endpoint_number – The endpoint number on which the host requested data.

static handle_generic_get_descriptor_request(*descriptor_container:* USBDevice | USBInterface, *request:* USBControlRequest)

Handle GET_DESCRIPTOR requests; per USB2 [9.4.3]

handle_get_supported_languages_descriptor() → bytes

Returns the special string-descriptor-zero that indicates which languages are supported.

handle_nak(*ep_num: int*)

Backend data-requested handler; for legacy compatibility.

Prefer overriding handle_data_requested() and handle_unexpected_data_Requested

handle_request(*request:* USBControlRequest)

Core control request handler.

This function can be overridden by a subclass if desired; but the typical way to handle a specific control request is to the the `@control_request_handler` decorators.

Parameters
request – the USBControlRequest object representing the relevant request

handle_unexpected_data_received(*endpoint_number: int*, *data: bytes*)

Handler for unexpected data.

Handles any data directed at an unexpected target; e.g. an endpoint that doesn't exist. Note that even if *handle_data_received* is overridden, this method can still be called e.g. by configuration.handle_data_received.

Parameters

- **endpoint_number** – The endpoint number on which the data was received.
- **data** – The raw bytes received on the relevant endpoint.

handle_unexpected_data_requested(*endpoint_number: int*)

Handler for unexpected data requests.

Handles any requests directed at an unexpected target; e.g. an endpoint that doesn't exist. Note that even if *handle_data_requested* is overridden, this method can still be called e.g. by configuration.handle_data_received.

Parameters
endpoint_number – The endpoint number on which the data was received.

manufacturer_string: str = 'FaceDancer'

max_packet_size_ep0: int = 64

name: str = 'generic device'

print_suggested_additions()

Prints a collection of suggested additions to the stdout.

product_id: int = 18003

product_string: str = 'Generic USB Device'

protocol_revision_number: int = 0

async run()

Runs the actual device emulation.

run_with(**coroutines: Iterable[Coroutine]*)

Runs the actual device emulation synchronously; running any provided coroutines simultaneously.

send(*endpoint_number: int*, *data: bytes*, ***, *blocking: bool = False*)

Queues sending data on the IN endpoint with the provided number.

Parameters

- **endpoint_number** – The endpoint number to send data upon.
- **data** – The data to send.
- **blocking** – If provided and true, this function will block until the backend indicates the send is complete.

serial_number_string: str = 'S/N 3420E'

set_address(*address: int*, *defer: bool = False*)

Updates the device's knowledge of its own address.

Parameters

- **address** – The address to apply.
- **defer** – If true, the address change should be deferred until the next time a control request ends. Should be set if we're changing the address before we ack the relevant transaction.

stall(***, *endpoint_number: int = 0*, *direction:* USBDirection *= USBDirection.OUT*)

Stalls the provided endpoint.

For endpoint zero, this indicates that the active (or next) request is not supported. For all other endpoints, this indicates a persistent 'halt' condition.

Parameters

endpoint – The endpoint address; or EP0 if not provided.

supported_languages: tuple = (LanguageIDs.ENGLISH_US,)

usb_spec_version: int = 2

vendor_id: int = 24843

class facedancer.device.**USBDevice**(*name: str = 'generic device'*, *device_class: int = 0*, *device_subclass: int = 0*, *protocol_revision_number: int = 0*, *max_packet_size_ep0: int = 64*, *vendor_id: int = 24843*, *product_id: int = 18003*, *manufacturer_string: str = 'FaceDancer'*, *product_string: str = 'Generic USB Device'*, *serial_number_string: str = 'S/N 3420E'*, *supported_languages: tuple = (LanguageIDs.ENGLISH_US*, *)*, *device_revision: int = 0*, *usb_spec_version: int = 2*, *device_speed: ~facedancer.types.DeviceSpeed = None*, *descriptors: ~typing.Dict[int*, *bytes | callable] = <factory>*, *configurations: ~typing.Dict[int*, *~facedancer.configuration.USBConfiguration] = <factory>*, *backend: ~facedancer.core.FacedancerUSBApp = None*)

Bases: *USBBaseDevice*

Class representing the behavior of a USB device.

This default implementation provides standard request handlers in order to facilitate creating a host-compatible USB device.

These functions can be overloaded to change their behavior. If you want to dramatically change the behavior of these requests, you can opt to use USBBaseDevice, which lacks standard request handling.

Fields:

device_class/device_subclass/protocol_revision_number –
The USB descriptor fields that select the class, subclass, and protcol.

vendor_id, product_id –
The USB vendor and product ID for this device.

manufacturer_string, product_string, serial_number_string –
Python strings identifying the device to the USB host.

supported_languages –
A tuple containing all of the language IDs supported by the device.

device_revision –
Number indicating the hardware revision of this device. Typically BCD.

usb_spec_revision –
Number indicating the version of the USB specification we adhere to. Typically 0x0200.

handle_clear_feature_request = <ControlRequestHandler wrapping USBDevice.handle_clear_feature_request at 0x7fae24d98140

handle_get_configuration_request = <ControlRequestHandler wrapping USBDevice.handle_get_configuration_request at 0x7fae24d9b0e0

handle_get_descriptor_request = <ControlRequestHandler wrapping USBDevice.handle_get_descriptor_request at 0x7fae24d9a570

handle_get_interface_request = <ControlRequestHandler wrapping USBDevice.handle_get_interface_request at 0x7fae24d9b650

handle_get_status_request = <ControlRequestHandler wrapping USBDevice.handle_get_status_request at 0x7fae24d9a240

handle_set_address_request = <ControlRequestHandler wrapping USBDevice.handle_set_address_request at 0x7fae24d9a210

handle_set_configuration_request = <ControlRequestHandler wrapping USBDevice.handle_set_configuration_request at 0x7fae24d9b3e0

handle_set_descriptor_request = <ControlRequestHandler wrapping USBDevice.handle_set_descriptor_request at 0x7fae24d9ae10

handle_set_feature_request = <ControlRequestHandler wrapping USBDevice.handle_set_feature_request at 0x7fae24d99df0

handle_set_interface_request = <ControlRequestHandler wrapping USBDevice.handle_set_interface_request at 0x7fae24d9b8c0

```
handle_synch_frame_request = <ControlRequestHandler wrapping
USBDevice.handle_synch_frame_request at 0x7fae24d9bb30
```

6.1.7 facedancer.endpoint module

Functionality for describing USB endpoints.

class facedancer.endpoint.**USBEndpoint**(*number: int*, *direction:* USBDirection, *transfer_type:* USBTransferType = *USBTransferType.BULK*, *synchronization_type:* USBSynchronizationType = *USBSynchronizationType.NONE*, *usage_type:* USBUsageType = *USBUsageType.DATA*, *max_packet_size: int = 64*, *interval: int = 0*, *parent:* USBDescribable = *None*)

Bases: *USBDescribable*, *AutoInstantiable*, *USBRequestHandler*

Class representing a USBEndpoint object.

Field:

number:
The endpoint number (without the direction bit) for this endpoint.

direction:
A USBDirection constant indicating this endpoint's direction.

transfer_type:
A USBTransferType constant indicating the type of communications used.

max_packet_size:
The maximum packet size for this endpoint.

interval:
The polling interval, for an INTERRUPT endpoint.

DESCRIPTOR_TYPE_NUMBER = 5

property address
Fetches the address for the given endpoint.

static address_for_number(*endpoint_number: int*, *direction:* USBDirection) → int
Computes the endpoint address for a given number + direction.

property attributes
Fetches the attributes for the given endpoint, as a single byte.

direction: *USBDirection*

classmethod from_binary_descriptor(*data*)
Creates an endpoint object from a description of that endpoint.

get_address()
Method alias for the address property. For backend support.

get_descriptor() → bytes
Get a descriptor string for this endpoint.

get_device()
Returns the device associated with the given descriptor.

get_identifier() → int

Returns a unique integer identifier for this object.

This is usually the index or address of the relevant USB object.

handle_buffer_empty()

Handler called when this endpoint first has an empty buffer.

handle_clear_feature_request = <ControlRequestHandler wrapping USBEndpoint.handle_clear_feature_request at 0x7fae24d99b20

handle_data_received(*data: bytes*)

Handler for receipt of non-control request data.

Parameters
data – The raw bytes received.

handle_data_requested()

Handler called when the host requests data on this endpoint.

interval: int = 0

matches_identifier(*other: int*) → bool

max_packet_size: int = 64

number: int

parent: *USBDescribable* = None

send(*data: bytes*, *, *blocking: bool = False*)

Sends data on this endpoint. Valid only for IN endpoints.

Parameters

- **data** – The data to be sent.
- **blocking** – True if we should block until the backend reports the transmission to be complete.

synchronization_type: *USBSynchronizationType* = 0

transfer_type: *USBTransferType* = 2

usage_type: *USBUsageType* = 0

6.1.8 facedancer.errors module

exception facedancer.errors.**DeviceNotFoundError**

Bases: OSError

Error indicating a device was not found.

6.1.9 facedancer.interface module

Functionality for defining USB interfaces.

class facedancer.interface.**USBInterface**(*name: str = 'generic USB interface'*, *number: int = 0*, *alternate: int = 0*, *class_number: int = 0*, *subclass_number: int = 0*, *protocol_number: int = 0*, *interface_string: str = None*, *descriptors: ~typing.Dict[int, bytes] = <factory>*, *class_descriptor: bytes = None*, *endpoints: ~typing.Dict[int, ~facedancer.endpoint.USBEndpoint] = <factory>*, *parent: ~facedancer.descriptor.USBDescribable = None*)

Bases: *USBDescribable*, *AutoInstantiable*, *USBRequestHandler*

Class representing a USBDevice interface.

Fields:

number :
The interface's index. Zero indexed.

class_number, subclass_number, protocol_number :
The USB class adhered to on this interface; usually a USBDeviceClass constant.

interface_string :
A short, descriptive string used to identify the endpoint; or None if not provided.

DESCRIPTOR_TYPE_NUMBER = 4

add_endpoint(*endpoint:* USBEndpoint)
Adds the provided endpoint to the interface.

alternate: int = 0

class_descriptor: bytes = None

class_number: int = 0

descriptors: Dict[int, bytes]

endpoints: Dict[int, *USBEndpoint*]

classmethod from_binary_descriptor(*data*)
Generates an interface object from a descriptor.

get_descriptor() → bytes
Retrieves the given interface's interface descriptor, with subordinates.

get_device()
Returns the device associated with the given descriptor.

get_endpoint(*endpoint_number: int*, *direction:* USBDirection) → *USBEndpoint*
Attempts to find a subordinate endpoint matching the given number/direction.

Parameters

- **endpoint_number** – The endpoint number to search for.
- **direction** – The endpoint direction to be matched.

Returns
The matching endpoint; or None if no matching endpoint existed.

get_endpoints()
Returns an iterable over all endpoints in this interface.

get_identifier() → int
Returns a unique integer identifier for this object.

This is usually the index or address of the relevant USB object.

handle_buffer_empty(*endpoint:* USBEndpoint)
Handler called when a given endpoint first has an empty buffer.

Often, an empty buffer indicates an opportunity to queue data for sending ('prime an endpoint'), but doesn't necessarily mean that the host is planning on reading the data.

This function is called only once per buffer.

handle_data_received(*endpoint:* USBEndpoint, *data: bytes*)
Handler for receipt of non-control request data.

Typically, this method will delegate any data received to the appropriate configuration/interface/endpoint. If overridden, the overriding function will receive all data; and can delegate it by calling the *.handle_data_received* method on *self.configuration*.

Parameters

- **endpoint_number** – The endpoint number on which the data was received.
- **data** – The raw bytes received on the relevant endpoint.

handle_data_requested(*endpoint:* USBEndpoint)
Handler called when the host requests data on a non-control endpoint.

Typically, this method will delegate the request to the appropriate interface+endpoint. If overridden, the overriding function will receive all data.

Parameters
endpoint_number – The endpoint number on which the host requested data.

handle_get_descriptor_request = <ControlRequestHandler wrapping USBInterface.handle_get_descriptor_request at 0x7fae24d99700>

handle_set_interface_request = <ControlRequestHandler wrapping USBInterface.handle_set_interface_request at 0x7fae24d992b0>

has_endpoint(*endpoint_number: int*, *direction:* USBDirection) → *USBEndpoint*
Returns true iff we have matching subordinate endpoint.

Parameters

- **endpoint_number** – The endpoint number to search for.
- **direction** – The endpoint direction to be matched.

interface_string: str = None

name: str = 'generic USB interface'

number: int = 0

parent: *USBDescribable* = None

protocol_number: int = 0

subclass_number: int = 0

6.1.10 facedancer.logging module

facedancer.logging.**configure_default_logging**(*level=20*, *logger=<module 'logging' from '/home/docs/.asdf/installs/python/3.12.0/lib/python3.12/logging/__init__.py'>*

6.1.11 facedancer.magic module

Functionally for automatic instantiations / tracking via decorators.

class facedancer.magic.**AutoInstantiable**

Bases: object

Base class for methods that can be decorated with use_automatically.

abstract get_identifier() → int

Returns a unique integer identifier for this object.

This is usually the index or address of the relevant USB object.

matches_identifier(*other: int*) → bool

class facedancer.magic.**AutoInstantiator**(*target_type*, *identifier=None*)

Bases: object

Simple wrapper class annotated on objects that can be instantiated automatically.

Used for the @use_automatically decorator; which removes a lot of the FaceDancer boilerplate at the cost of being somewhat cryptic.

creates_instance_of(*expected_type*)

facedancer.magic.**instantiate_subordinates**(*obj*, *expected_type*)

Automatically instantiates any inner classes with a matching type.

This is used by objects that represent USB hardware behaviors (e.g. USBDevice, USBConfiguration, USBInterface, USBEndpoint) in order to automatically create objects of any inner class decorated with `@use_automatically`.

facedancer.magic.**use_automatically**(*cls*)

Class decorator used to annotate FaceDancer inner classes. Implies @dataclass.

This decorator can be placed on inner classes that describe "subordinate" objects on USB devices. For example, a USBDevice can have several subordinate USBConfigurations; which select the various configurations for that class.

When placed on a subordinate class, this allows the parent class to automatically instantiate the relevant given class during its creation; automatically populating the subordinate properties of the relevant device.

For example, assume we have a FaceDancer class representing a custom USB device:

```
@dataclass
class ExampleDevice(USBDevice):
    product_string : str = "My Example Device"

    @use_automatically
    class DefaultConfiguration(USBConfiguration):
        number : int = 1
```

In this case, when an ExampleDevice is instantiated, the USBDevice code knows how to instantiate DefaultConfiguration, and will do so automatically.

Note that this decorator should _only_ be used for subordinate types; and expects that the decorated class has no explicitly-declared __init__ method. The __post_init__ mechanism of python dataclasses can be overridden to perform any needed initialization.

facedancer.magic.**use_inner_classes_automatically**(*cls*)

Decorator that acts as if all inner classes were defined with *use_automatically*.

6.1.12 facedancer.proxy module

USB Proxy implementation.

class facedancer.proxy.**LibUSB1Device**

Bases: object

A wrapper around the proxied device based on libusb1.

context = None

Class variable that stores our device handle.

classmethod controlRead(*request_type*, *request*, *value*, *index*, *length*, *timeout=1000*)

classmethod controlWrite(*request_type*, *request*, *value*, *index*, *data*, *timeout=1000*)

device_handle = None

classmethod device_speed()

classmethod find(*idVendor*, *idProduct*, *find_all=True*)

Finds a USB device by its identifiers.

classmethod open(*device*, *detach=True*)

classmethod read(*endpoint_number*, *length*, *timeout=1000*)

classmethod write(*endpoint_number*, *data*, *timeout=1000*)

class facedancer.proxy.**USBProxyDevice**(*index=0*, *quirks=[]*, *scheduler=None*, ***kwargs*)

Bases: *USBBaseDevice*

USB Proxy Device

__init__(*index=0*, *quirks=[]*, *scheduler=None*, ***kwargs*)

Sets up a new USBProxy instance.

add_filter(*filter_object*, *head=False*)

Adds a filter to the USBProxy filter stack.

configured(*configuration:* USBConfiguration)

Callback that handles when the target device becomes configured. If you're using the standard filters, this will be called automatically; if not, you'll have to call it once you know the device has been configured.

Parameters
configuration – The configuration to be applied.

connect()

Initialize this device. We perform a reduced initialization, as we really only want to proxy data.

filter_list = []

handle_bus_reset()

Event handler for a bus reset.

handle_data_available(*ep_num*, *data*)

Handles the case where data is ready from the Facedancer device that needs to be proxied to the target device.

handle_get_configuration_request(*request*)

handle_get_descriptor_request(*request*)

handle_nak(*ep_num*)

Handles a NAK, which means that the target asked the proxied device to participate in a transfer. We use this as our cue to participate in communications.

handle_request(*request: USBControlRequest*)

Proxies EP0 requests between the victim and the target.

name: str = 'USB Proxy Device'

6.1.13 facedancer.request module

Functionality for declaring and working with USB control requests.

class facedancer.request.**ControlRequestHandler**(*handler_function*, *execution_condition*)

Bases: object

Class representing a control request handler.

Instances of this class are generated automatically each time a control request is defined using decorator syntax; and track the association between the relevant handler function and the condition under which it's executed.

__call__(*caller*, *request*)

Primary execution; calls the relevant handler if our conditions are met.

add_condition(*condition*)

Refines a control request handler such that it's only called when the added condition is true.

add_field_matcher(*field_name*, *field_value*)

Refines a control request handler such that it's only called when one of its fields matches a given value.

Parameters

- **field_name** – The property of the USBControlRequest object to be checked.
- **field_value** – The value the relevant property must match to be called.

class facedancer.request.**USBControlRequest**(*direction: USBDirection*, *type: USBRequestType*, *recipient: USBRequestRecipient*, *number: int*, *value: int*, *index: int*, *length: int*, *data: bytes = b''*, *device: USBDescribable = None*)

Bases: object

Class encapsulating a USB control request.

TODO: document parameters

ack(*, *blocking: bool = False*)

Acknowledge the given request without replying.

Convenience alias for .acknowledge().

Parameters
blocking – If true, the relevant control request will complete before returning.

acknowledge(*, *blocking: bool = False*)

Acknowledge the given request without replying.

Parameters
blocking – If true, the relevant control request will complete before returning.

data: bytes = b''

device: *USBDescribable* = None

direction: *USBDirection*

classmethod from_raw_bytes(*raw_bytes: bytes*, *, *device=None*)

Creates a request object from a sequence of raw bytes.

Parameters

- **raw_bytes** – The raw bytes to create the object from.
- **device** – The USBDevice to associate with the given request. Optional, but necessary to use the .reply() / .acknowledge() methods.

get_direction() → *USBDirection*

get_recipient() → *USBRequestRecipient*

get_type() → *USBRequestType*

index: int

property index_high: int

property index_low: int

length: int

number: int

raw() → bytes

Returns the raw bytes that compose the request.

recipient: *USBRequestRecipient*

reply(*data: bytes*)

Replies to the given request with a given set of bytes.

property request: int

property request_type: int

Fetches the whole *request_type* byte.

stall()

Stalls the associated device's control request.

Used to indicate that a given request isn't supported; or isn't supported with the provided arguments.

type: *USBRequestType*

value: int

property value_high: int

property value_low: int

class facedancer.request.**USBRequestHandler**

Bases: object

Base class for any object that handles USB requests.

handle_request(*request:* USBControlRequest) → bool

Core control request handler.

This function can be overridden by a subclass if desired; but the typical way to handle a specific control request is to the the @control_request_handler decorators.

Parameters
request – the USBControlRequest object representing the relevant request

Returns
true iff the request is handled

facedancer.request.**class_request_handler**(***kwargs*)

Decorator; declares a class request handler. See control_request_handler() for usage.

facedancer.request.**control_request_handler**(*condition=<function <lambda>>*, ***kwargs*)

Decorator that declares a control request handler.

Used while defining a USBDevice, USBInterface, USBEndpoint, or USBOtherRecipient class to declare handlers for that function.

Parameters
condition – A function that, when evaluated on a USBControlRequest, evaluates true if and only if this function is an appropriate handler.

facedancer.request.**get_request_handler_methods**(*cls*) → List[callable]

Returns a list of all handler methods on a given class or object.

This is used to find all methods of an object decorated with the @*_request_handler decorators.

facedancer.request.**reserved_request_handler**(***kwargs*)

Decorator; declares a reserved-type request handler. Not typically used.

facedancer.request.**standard_request_handler**(***kwargs*)

Decorator; declares a standard request handler. See control_request_handler() for usage.

facedancer.request.**to_any_endpoint**(*func*)

Decorator; refines a handler so it's only called on requests with an endpoint recipient.

facedancer.request.**to_any_interface**(*func*)

Decorator; refines a handler so it's only called on requests with an interface recipient.

facedancer.request.**to_device**(*func*)

Decorator; refines a handler so it's only called on requests with a device recipient.

facedancer.request.**to_other**(*func*)

Decorator; refines a handler so it's only called on requests with an Other (TM) recipient.

facedancer.request.**to_this_endpoint**(*func*)

Decorator; refines a handler so it's only called on requests targeting this endpoint.

facedancer.request.**to_this_interface**(*func*)

Decorator; refines a handler so it's only called on requests targeting this interface.

facedancer.request.**vendor_request_handler**(***kwargs*)

Decorator; declares a vendor request handler. See control_request_handler() for usage.

6.1.14 facedancer.types module

USB types – defines enumerations that describe standard USB types

class facedancer.types.**DescriptorTypes**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

CONFIGURATION = 2

DEVICE = 1

DEVICE_QUALIFIER = 6

ENDPOINT = 5

HID = 33

INTERFACE = 4

INTERFACE_POWER = 8

OTHER_SPEED_CONFIGURATION = 7

REPORT = 34

STRING = 3

class facedancer.types.**DeviceSpeed**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

FULL = 2

HIGH = 3

LOW = 1

SUPER = 4

SUPER_PLUS = 5

UNKNOWN = 0

class facedancer.types.**LanguageIDs**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

AFRIKAANS = 1078

ALBANIAN = 1052

ARABIC_ALGERIA = 5121

ARABIC_BAHRAIN = 15361

ARABIC_EGYPT = 3073

ARABIC_IRAQ = 2049

ARABIC_JORDAN = 11265

ARABIC_KUWAIT = 13313

ARABIC_LEBANON = 12289

ARABIC_LIBYA = 4097

ARABIC_MOROCCO = 6145

ARABIC_OMAN = 8193

ARABIC_QATAR = 16385

ARABIC_SAUDI_ARABIA = 1025

ARABIC_SYRIA = 10241

ARABIC_TUNISIA = 7169

ARABIC_UAE = 14337

ARABIC_YEMEN = 9217

ARMENIAN = 1067

ASSAMESE = 1101

AZERI_CYRILLIC = 2092

AZERI_LATIN = 1068

BASQUE = 1069

BELARUSSIAN = 1059

BENGALI = 1093

BULGARIAN = 1026

BURMESE = 1109

CATALAN = 1027

```
CHINESE_HONG_KONG = 3076

CHINESE_MACAU_SAR = 5124

CHINESE_PRC = 2052

CHINESE_SINGAPORE = 4100

CHINESE_TAIWAN = 1028

CROATIAN = 1050

CZECH = 1029

DANISH = 1030

DUTCH_BELGIUM = 2067

DUTCH_NETHERLANDS = 1043

ENGLISH_AUSTRALIAN = 3081

ENGLISH_BELIZE = 10249

ENGLISH_CANADIAN = 4105

ENGLISH_CARIBBEAN = 9225

ENGLISH_IRELAND = 6153

ENGLISH_JAMAICA = 8201

ENGLISH_NEW_ZEALAND = 5129

ENGLISH_PHILIPPINES = 13321

ENGLISH_SOUTH_AFRICA = 7177

ENGLISH_TRINIDAD = 11273

ENGLISH_UNITED_KINGDOM = 2057

ENGLISH_US = 1033

ENGLISH_ZIMBABWE = 12297

ESTONIAN = 1061

FAEROESE = 1080

FARSI = 1065

FINNISH = 1035

FRENCH_BELGIAN = 2060

FRENCH_CANADIAN = 3084

FRENCH_LUXEMBOURG = 5132

FRENCH_MONACO = 6156
```

FRENCH_STANDARD = 1036

FRENCH_SWITZERLAND = 4108

GEORGIAN = 1079

GERMAN_AUSTRIA = 3079

GERMAN_LIECHTENSTEIN = 5127

GERMAN_LUXEMBOURG = 4103

GERMAN_STANDARD = 1031

GERMAN_SWITZERLAND = 2055

GREEK = 1032

GUJARATI = 1095

HEBREW = 1037

HID_USAGE_DATA_DESCRIPTOR = 1279

HID_VENDOR_DEFINED_1 = 61695

HID_VENDOR_DEFINED_2 = 62719

HID_VENDOR_DEFINED_3 = 63743

HID_VENDOR_DEFINED_4 = 64767

HINDI = 1081

HUNGARIAN = 1038

ICELANDIC = 1039

INDONESIAN = 1057

ITALIAN_STANDARD = 1040

ITALIAN_SWITZERLAND = 2064

JAPANESE = 1041

KANNADA = 1099

KASHMIRI_INDIA = 2144

KAZAKH = 1087

KONKANI = 1111

KOREAN = 1042

KOREAN_JOHAB = 2066

LATVIAN = 1062

LITHUANIAN = 1063

```
    LITHUANIAN_CLASSIC = 2087

    MACEDONIAN = 1071

    MALAYALAM = 1100

    MALAY_BRUNEI_DARUSSALAM = 2110

    MALAY_MALAYSIAN = 1086

    MANIPURI = 1112

    MARATHI = 1102

    NEPALI_INDIA = 2145

    NORWEGIAN_BOKMAL = 1044

    NORWEGIAN_NYNORSK = 2068

    ORIYA = 1096

    POLISH = 1045

    PORTUGUESE_BRAZIL = 1046

    PORTUGUESE_STANDARD = 2070

    PUNJABI = 1094

    ROMANIAN = 1048

    RUSSIAN = 1049

    SANSKRIT = 1103

    SERBIAN_CYRILLIC = 3098

    SERBIAN_LATIN = 2074

    SINDHI = 1113

    SLOVAK = 1051

    SLOVENIAN = 1060

    SPANISH_ARGENTINA = 11274

    SPANISH_BOLIVIA = 16394

    SPANISH_CHILE = 13322

    SPANISH_COLOMBIA = 9226

    SPANISH_COSTA_RICA = 5130

    SPANISH_DOMINICAN_REPUBLIC = 7178

    SPANISH_ECUADOR = 12298

    SPANISH_EL_SALVADOR = 17418
```

SPANISH_GUATEMALA = 4106

SPANISH_HONDURAS = 18442

SPANISH_MEXICAN = 2058

SPANISH_MODERN_SORT = 3082

SPANISH_NICARAGUA = 19466

SPANISH_PANAMA = 6154

SPANISH_PARAGUAY = 15370

SPANISH_PERU = 10250

SPANISH_PUERTO_RICO = 20490

SPANISH_TRADITIONAL_SORT = 1034

SPANISH_URUGUAY = 14346

SPANISH_VENEZUELA = 8202

SUTU = 1072

SWAHILI_KENYA = 1089

SWEDISH = 1053

SWEDISH_FINLAND = 2077

TAMIL = 1097

TATAR_TATARSTAN = 1092

TELUGU = 1098

THAI = 1054

TURKISH = 1055

UKRAINIAN = 1058

URDU_INDIA = 2080

URDU_PAKISTAN = 1056

UZBEK_CYRILLIC = 2115

UZBEK_LATIN = 1091

VIETNAMESE = 1066

class facedancer.types.USB

Bases: object

desc_type_configuration = 2

desc_type_device = 1

desc_type_device_qualifier = 6

desc_type_endpoint = 5

desc_type_hid = 33

desc_type_interface = 4

desc_type_interface_power = 8

desc_type_other_speed_configuration = 7

desc_type_report = 34

desc_type_string = 3

feature_device_remote_wakeup = 1

feature_endpoint_halt = 0

feature_test_mode = 2

if_class_to_desc_type = {3: 33}

interface_class_to_descriptor_type()

request_direction_device_to_host = 1

request_direction_host_to_device = 0

request_recipient_device = 0

request_recipient_endpoint = 2

request_recipient_interface = 1

request_recipient_other = 3

request_type_class = 1

request_type_standard = 0

request_type_vendor = 2

state_address = 4

state_attached = 1

state_configured = 5

state_default = 3

state_detached = 0

state_powered = 2

state_suspended = 6

class facedancer.types.**USBDirection**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

Class representing USB directions.

IN = 1

OUT = 0

classmethod from_endpoint_address(*address*)

Helper method that extracts the direction from an endpoint address.

classmethod from_request_type(*request_type_int*)

Helper method that extracts the direction from a request_type integer.

is_in()

is_out()

classmethod parse(*value*)

Helper that converts a numeric field into a direction.

reverse()

Returns the reverse of the given direction.

to_endpoint_address(*endpoint_number*)

Helper method that converts and endpoint_number to an address, given direction.

token()

Generates the token corresponding to the given direction.

class facedancer.types.**USBPIDCategory**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntFlag

Category constants for each of the groups that PIDs can fall under.

DATA = 3

HANDSHAKE = 2

MASK = 3

SPECIAL = 0

TOKEN = 1

class facedancer.types.**USBPacketID**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntFlag

Enumeration specifying all of the valid USB PIDs we can handle.

ACK = 2

DATA0 = 3

DATA1 = 11

DATA2 = 7

ERR = 12

IN = 9

MDATA = 15

NAK = 10

NYET = 6

OUT = 1

PID_CORE_MASK = 15

PID_INVALID = 16

PING = 4

PRE = 12

SETUP = 13

SOF = 5

SPLIT = 8

STALL = 14

category()

Returns the USBPIDCategory that each given PID belongs to.

direction()

Get a USB direction from a PacketID.

classmethod from_byte(*byte*, *skip_checks=False*)

Creates a PID object from a byte.

classmethod from_int(*value*, *skip_checks=True*)

Create a PID object from an integer.

classmethod from_name(*name*)

Create a PID object from a string representation of its name.

is_data()

Returns true iff the given PID represents a DATA packet.

is_handshake()

Returns true iff the given PID represents a handshake packet.

is_invalid()

Returns true if this object is an attempt to encapsulate an invalid PID.

is_token()

Returns true iff the given PID represents a token packet.

classmethod parse(*value*)

Attempt to create a PID object from a number, byte, or string.

summarize()

Return a summary of the given packet.

class facedancer.types.**USBRequestRecipient**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

Enumeration that describes each 'recipient' of a USB request field.

DEVICE = 0

ENDPOINT = 2

INTERFACE = 1

OTHER = 3

RESERVED = 4

classmethod from_integer(*value*)

Special factory that correctly handles reserved values.

classmethod from_request_type(*request_type_int*)

Helper method that extracts the type from a request_type integer.

class facedancer.types.**USBRequestType**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

Enumeration that describes each possible Type field for a USB request.

CLASS = 1

RESERVED = 3

STANDARD = 0

VENDOR = 2

classmethod from_request_type(*request_type_int*)

Helper method that extracts the type from a request_type integer.

class facedancer.types.**USBStandardRequests**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

CLEAR_FEATURE = 1

GET_CONFIGURATION = 8

GET_DESCRIPTOR = 6

GET_INTERFACE = 10

GET_STATUS = 0

SET_ADDRESS = 5

SET_CONFIGURATION = 9

SET_DESCRIPTOR = 7

SET_FEATURE = 3

SET_INTERFACE = 11

SYNCH_FRAME = 12

class facedancer.types.**USBSynchronizationType**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

ADAPTIVE = 2

ASYNC = 1

NONE = 0

SYNCHRONOUS = 3

class facedancer.types.**USBTransferType**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

BULK = 2

CONTROL = 0

INTERRUPT = 3

ISOCHRONOUS = 1

class facedancer.types.**USBUsageType**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

DATA = 0

FEEDBACK = 1

IMPLICIT_FEEDBACK = 2

facedancer.types.**endpoint_number_from_address**(*number*)

6.1.15 Module contents

class facedancer.**DeviceSpeed**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

FULL = 2

HIGH = 3

LOW = 1

SUPER = 4

SUPER_PLUS = 5

UNKNOWN = 0

class facedancer.**LanguageIDs**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

AFRIKAANS = 1078

ALBANIAN = 1052

ARABIC_ALGERIA = 5121

ARABIC_BAHRAIN = 15361

ARABIC_EGYPT = 3073

ARABIC_IRAQ = 2049

ARABIC_JORDAN = 11265

ARABIC_KUWAIT = 13313

ARABIC_LEBANON = 12289

ARABIC_LIBYA = 4097

ARABIC_MOROCCO = 6145

ARABIC_OMAN = 8193

ARABIC_QATAR = 16385

ARABIC_SAUDI_ARABIA = 1025

ARABIC_SYRIA = 10241

ARABIC_TUNISIA = 7169

ARABIC_UAE = 14337

ARABIC_YEMEN = 9217

ARMENIAN = 1067

ASSAMESE = 1101

AZERI_CYRILLIC = 2092

AZERI_LATIN = 1068

BASQUE = 1069

BELARUSSIAN = 1059

BENGALI = 1093

BULGARIAN = 1026

```
BURMESE = 1109

CATALAN = 1027

CHINESE_HONG_KONG = 3076

CHINESE_MACAU_SAR = 5124

CHINESE_PRC = 2052

CHINESE_SINGAPORE = 4100

CHINESE_TAIWAN = 1028

CROATIAN = 1050

CZECH = 1029

DANISH = 1030

DUTCH_BELGIUM = 2067

DUTCH_NETHERLANDS = 1043

ENGLISH_AUSTRALIAN = 3081

ENGLISH_BELIZE = 10249

ENGLISH_CANADIAN = 4105

ENGLISH_CARIBBEAN = 9225

ENGLISH_IRELAND = 6153

ENGLISH_JAMAICA = 8201

ENGLISH_NEW_ZEALAND = 5129

ENGLISH_PHILIPPINES = 13321

ENGLISH_SOUTH_AFRICA = 7177

ENGLISH_TRINIDAD = 11273

ENGLISH_UNITED_KINGDOM = 2057

ENGLISH_US = 1033

ENGLISH_ZIMBABWE = 12297

ESTONIAN = 1061

FAEROESE = 1080

FARSI = 1065

FINNISH = 1035

FRENCH_BELGIAN = 2060

FRENCH_CANADIAN = 3084
```

FRENCH_LUXEMBOURG = 5132

FRENCH_MONACO = 6156

FRENCH_STANDARD = 1036

FRENCH_SWITZERLAND = 4108

GEORGIAN = 1079

GERMAN_AUSTRIA = 3079

GERMAN_LIECHTENSTEIN = 5127

GERMAN_LUXEMBOURG = 4103

GERMAN_STANDARD = 1031

GERMAN_SWITZERLAND = 2055

GREEK = 1032

GUJARATI = 1095

HEBREW = 1037

HID_USAGE_DATA_DESCRIPTOR = 1279

HID_VENDOR_DEFINED_1 = 61695

HID_VENDOR_DEFINED_2 = 62719

HID_VENDOR_DEFINED_3 = 63743

HID_VENDOR_DEFINED_4 = 64767

HINDI = 1081

HUNGARIAN = 1038

ICELANDIC = 1039

INDONESIAN = 1057

ITALIAN_STANDARD = 1040

ITALIAN_SWITZERLAND = 2064

JAPANESE = 1041

KANNADA = 1099

KASHMIRI_INDIA = 2144

KAZAKH = 1087

KONKANI = 1111

KOREAN = 1042

KOREAN_JOHAB = 2066

```
LATVIAN = 1062

LITHUANIAN = 1063

LITHUANIAN_CLASSIC = 2087

MACEDONIAN = 1071

MALAYALAM = 1100

MALAY_BRUNEI_DARUSSALAM = 2110

MALAY_MALAYSIAN = 1086

MANIPURI = 1112

MARATHI = 1102

NEPALI_INDIA = 2145

NORWEGIAN_BOKMAL = 1044

NORWEGIAN_NYNORSK = 2068

ORIYA = 1096

POLISH = 1045

PORTUGUESE_BRAZIL = 1046

PORTUGUESE_STANDARD = 2070

PUNJABI = 1094

ROMANIAN = 1048

RUSSIAN = 1049

SANSKRIT = 1103

SERBIAN_CYRILLIC = 3098

SERBIAN_LATIN = 2074

SINDHI = 1113

SLOVAK = 1051

SLOVENIAN = 1060

SPANISH_ARGENTINA = 11274

SPANISH_BOLIVIA = 16394

SPANISH_CHILE = 13322

SPANISH_COLOMBIA = 9226

SPANISH_COSTA_RICA = 5130

SPANISH_DOMINICAN_REPUBLIC = 7178
```

SPANISH_ECUADOR = 12298

SPANISH_EL_SALVADOR = 17418

SPANISH_GUATEMALA = 4106

SPANISH_HONDURAS = 18442

SPANISH_MEXICAN = 2058

SPANISH_MODERN_SORT = 3082

SPANISH_NICARAGUA = 19466

SPANISH_PANAMA = 6154

SPANISH_PARAGUAY = 15370

SPANISH_PERU = 10250

SPANISH_PUERTO_RICO = 20490

SPANISH_TRADITIONAL_SORT = 1034

SPANISH_URUGUAY = 14346

SPANISH_VENEZUELA = 8202

SUTU = 1072

SWAHILI_KENYA = 1089

SWEDISH = 1053

SWEDISH_FINLAND = 2077

TAMIL = 1097

TATAR_TATARSTAN = 1092

TELUGU = 1098

THAI = 1054

TURKISH = 1055

UKRAINIAN = 1058

URDU_INDIA = 2080

URDU_PAKISTAN = 1056

UZBEK_CYRILLIC = 2115

UZBEK_LATIN = 1091

VIETNAMESE = 1066

class facedancer.**USBClassDescriptor**(*number: int*, *raw: bytes*, *type_number: int = None*, *parent:* USBDescribable *= None*)

Bases: *USBDescriptor*

Class for arbitrary USB Class descriptors.

class facedancer.**USBConfiguration**(*number: int = 1*, *configuration_string: str = None*, *max_power: int = 500*, *self_powered: bool = True*, *supports_remote_wakeup: bool = True*, *parent: ~facedancer.descriptor.USBDescribable = None*, *interfaces: ~facedancer.interface.USBInterface =* <*factory*>)

Bases: *USBDescribable*, *AutoInstantiable*, *USBRequestHandler*

Class representing a USBDevice's configuration.

Fields:

number:
The configuration's number; one-indexed.

configuration_string
A string describing the configuration; or None if not provided.

max_power:
The maximum power expected to be drawn by the device when using this interface, in mA. Typically 500mA, for maximum possible.

supports_remote_wakeup:
True iff this device should be able to wake the host from suspend.

DESCRIPTOR_SIZE_BYTES = 9

DESCRIPTOR_TYPE_NUMBER = 2

add_interface(*interface:* USBInterface)

Adds an interface to the configuration.

property attributes

Retrives the "attributes" composite word.

configuration_string: str = None

classmethod from_binary_descriptor(*data*)

Generates a new USBConfiguration object from a configuration descriptor, handling any attached subordinate descriptors.

Parameters
data – The raw bytes for the descriptor to be parsed.

get_descriptor() → bytes

Returns this configuration's configuration descriptor, including subordinates.

get_device()

Returns a reference to the associated device.

get_endpoint(*number: int*, *direction:* USBDirection) → *USBEndpoint*

Attempts to find an endpoint with the given number + direction.

Parameters

- **number** – The endpoint number to look for.
- **direction** – Whether to look for an IN or OUT endpoint.

get_identifier() → int

Returns a unique integer identifier for this object.

This is usually the index or address of the relevant USB object.

get_interfaces() → Iterable[*USBInterface*]

Returns an iterable over all interfaces on the provided device.

handle_buffer_empty(*endpoint:* USBEndpoint)

Handler called when a given endpoint first has an empty buffer.

Often, an empty buffer indicates an opportunity to queue data for sending ('prime an endpoint'), but doesn't necessarily mean that the host is planning on reading the data.

This function is called only once per buffer.

handle_data_received(*endpoint:* USBEndpoint, *data: bytes*)

Handler for receipt of non-control request data.

Typically, this method will delegate any data received to the appropriate configuration/interface/endpoint. If overridden, the overriding function will receive all data; and can delegate it by calling the *.handle_data_received* method on *self.configuration*.

Parameters

- **endpoint** – The endpoint on which the data was received.
- **data** – The raw bytes received on the relevant endpoint.

handle_data_requested(*endpoint:* USBEndpoint)

Handler called when the host requests data on a non-control endpoint.

Typically, this method will delegate the request to the appropriate interface+endpoint. If overridden, the overriding function will receive all data.

Parameters

endpoint – The endpoint on which the host requested data.

interfaces: *USBInterface*

max_power: int = 500

number: int = 1

parent: *USBDescribable* = None

self_powered: bool = True

supports_remote_wakeup: bool = True

class facedancer.**USBControlRequest**(*direction:* USBDirection, *type:* USBRequestType, *recipient:* USBRequestRecipient, *number: int*, *value: int*, *index: int*, *length: int*, *data: bytes = b''*, *device:* USBDescribable = *None*)

Bases: object

Class encapsulating a USB control request.

TODO: document parameters

ack(*, *blocking: bool = False*)

Acknowledge the given request without replying.

Convenience alias for .acknowledge().

Parameters
blocking – If true, the relevant control request will complete before returning.

acknowledge(*, *blocking: bool = False*)

Acknowledge the given request without replying.

Parameters
blocking – If true, the relevant control request will complete before returning.

data: bytes = b''

device: *USBDescribable* = None

direction: *USBDirection*

classmethod from_raw_bytes(*raw_bytes: bytes*, *, *device=None*)

Creates a request object from a sequence of raw bytes.

Parameters

- **raw_bytes** – The raw bytes to create the object from.
- **device** – The USBDevice to associate with the given request. Optional, but necessary to use the .reply() / .acknowledge() methods.

get_direction() → *USBDirection*

get_recipient() → *USBRequestRecipient*

get_type() → *USBRequestType*

index: int

property index_high: int

property index_low: int

length: int

number: int

raw() → bytes

Returns the raw bytes that compose the request.

recipient: *USBRequestRecipient*

reply(*data: bytes*)

Replies to the given request with a given set of bytes.

property request: int

property request_type: int

Fetches the whole *request_type* byte.

stall()

Stalls the associated device's control request.

Used to indicate that a given request isn't supported; or isn't supported with the provided arguments.

type: *USBRequestType*

value: int

property value_high: int

property value_low: int

class facedancer.**USBDescriptor**(*number: int*, *raw: bytes*, *type_number: int = None*, *parent: USBDescribable = None*)

Bases: *USBDescribable*, *AutoInstantiable*

Class for arbitrary USB descriptors; minimal concrete implementation of USBDescribable.

__call__(*index=0*)

Converts the descriptor object into raw bytes.

get_identifier()

Returns a unique integer identifier for this object.

This is usually the index or address of the relevant USB object.

number: int

parent: *USBDescribable* = None

raw: bytes

type_number: int = None

class facedancer.**USBDescriptorTypeNumber**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

CONFIGURATION = 2

DEVICE = 1

DEVICE_QUALIFIER = 6

ENDPOINT = 5

HID = 33

INTERFACE = 4

INTERFACE_POWER = 8

OTHER_SPEED_CONFIGURATION = 7

REPORT = 34

STRING = 3

class facedancer.**USBDevice**(*name: str = 'generic device'*, *device_class: int = 0*, *device_subclass: int = 0*, *protocol_revision_number: int = 0*, *max_packet_size_ep0: int = 64*, *vendor_id: int = 24843*, *product_id: int = 18003*, *manufacturer_string: str = 'FaceDancer'*, *product_string: str = 'Generic USB Device'*, *serial_number_string: str = 'S/N 3420E'*, *supported_languages: tuple = (LanguageIDs.ENGLISH_US,)*, *device_revision: int = 0*, *usb_spec_version: int = 2*, *device_speed: ~facedancer.types.DeviceSpeed = None*, *descriptors: ~typing.Dict[int, bytes | callable] = <factory>*, *configurations: ~typing.Dict[int, ~facedancer.configuration.USBConfiguration] = <factory>*, *backend: ~facedancer.core.FacedancerUSBApp = None*)

Bases: *USBBaseDevice*

Class representing the behavior of a USB device.

This default implementation provides standard request handlers in order to facilitate creating a host-compatible USB device.

These functions can be overloaded to change their behavior. If you want to dramatically change the behavior of these requests, you can opt to use USBBaseDevice, which lacks standard request handling.

Fields:

device_class/device_subclass/protocol_revision_number –
The USB descriptor fields that select the class, subclass, and protcol.

vendor_id, product_id –
The USB vendor and product ID for this device.

manufacturer_string, product_string, serial_number_string –
Python strings identifying the device to the USB host.

supported_languages –
A tuple containing all of the language IDs supported by the device.

device_revision –
Number indicating the hardware revision of this device. Typically BCD.

usb_spec_revision –
Number indicating the version of the USB specification we adhere to. Typically 0x0200.

configurations: Dict[int, *USBConfiguration*]

descriptors: Dict[int, bytes | callable]

handle_clear_feature_request = <ControlRequestHandler wrapping USBDevice.handle_clear_feature_request at 0x7fae24d98140

handle_get_configuration_request = <ControlRequestHandler wrapping USBDevice.handle_get_configuration_request at 0x7fae24d9b0e0

handle_get_descriptor_request = <ControlRequestHandler wrapping USBDevice.handle_get_descriptor_request at 0x7fae24d9a570

handle_get_interface_request = <ControlRequestHandler wrapping USBDevice.handle_get_interface_request at 0x7fae24d9b650

handle_get_status_request = <ControlRequestHandler wrapping USBDevice.handle_get_status_request at 0x7fae24d9a240

handle_set_address_request = <ControlRequestHandler wrapping USBDevice.handle_set_address_request at 0x7fae24d9a210

handle_set_configuration_request = <ControlRequestHandler wrapping USBDevice.handle_set_configuration_request at 0x7fae24d9b3e0

handle_set_descriptor_request = <ControlRequestHandler wrapping USBDevice.handle_set_descriptor_request at 0x7fae24d9ae10

handle_set_feature_request = <ControlRequestHandler wrapping USBDevice.handle_set_feature_request at 0x7fae24d99df0

handle_set_interface_request = <ControlRequestHandler wrapping USBDevice.handle_set_interface_request at 0x7fae24d9b8c0

handle_synch_frame_request = <ControlRequestHandler wrapping USBDevice.handle_synch_frame_request at 0x7fae24d9bb30

class facedancer.**USBDirection**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

Class representing USB directions.

IN = 1

OUT = 0

classmethod from_endpoint_address(*address*)

Helper method that extracts the direction from an endpoint address.

classmethod from_request_type(*request_type_int*)

Helper method that extracts the direction from a request_type integer.

is_in()

is_out()

classmethod parse(*value*)

Helper that converts a numeric field into a direction.

reverse()

Returns the reverse of the given direction.

to_endpoint_address(*endpoint_number*)

Helper method that converts and endpoint_number to an address, given direction.

token()

Generates the token corresponding to the given direction.

class facedancer.**USBEndpoint**(*number: int*, *direction:* USBDirection, *transfer_type:* USBTransferType = *USBTransferType.BULK*, *synchronization_type:* USBSynchronizationType = *USBSynchronizationType.NONE*, *usage_type:* USBUsageType = *USBUsageType.DATA*, *max_packet_size: int = 64*, *interval: int = 0*, *parent:* USBDescribable = *None*)

Bases: *USBDescribable*, *AutoInstantiable*, *USBRequestHandler*

Class representing a USBEndpoint object.

Field:

number:
The endpoint number (without the direction bit) for this endpoint.

direction:
A USBDirection constant indicating this endpoint's direction.

transfer_type:
A USBTransferType constant indicating the type of communications used.

max_packet_size:
The maximum packet size for this endpoint.

interval:
The polling interval, for an INTERRUPT endpoint.

DESCRIPTOR_TYPE_NUMBER = 5

property address
Fetches the address for the given endpoint.

static address_for_number(*endpoint_number: int*, *direction:* USBDirection) → int
Computes the endpoint address for a given number + direction.

property attributes
Fetches the attributes for the given endpoint, as a single byte.

direction: *USBDirection*

classmethod from_binary_descriptor(*data*)
Creates an endpoint object from a description of that endpoint.

get_address()
Method alias for the address property. For backend support.

get_descriptor() → bytes
Get a descriptor string for this endpoint.

get_device()
Returns the device associated with the given descriptor.

get_identifier() → int
Returns a unique integer identifier for this object.

This is usually the index or address of the relevant USB object.

handle_buffer_empty()
Handler called when this endpoint first has an empty buffer.

handle_clear_feature_request = <ControlRequestHandler wrapping USBEndpoint.handle_clear_feature_request at 0x7fae24d99b20

handle_data_received(*data: bytes*)
Handler for receipt of non-control request data.

Parameters
data – The raw bytes received.

handle_data_requested()
Handler called when the host requests data on this endpoint.

interval: `int = 0`

matches_identifier(*other: int*) → bool

max_packet_size: `int = 64`

number: `int`

parent: *USBDescribable* `= None`

send(*data: bytes*, *, *blocking: bool = False*)

Sends data on this endpoint. Valid only for IN endpoints.

Parameters

- **data** – The data to be sent.
- **blocking** – True if we should block until the backend reports the transmission to be complete.

synchronization_type: *USBSynchronizationType* `= 0`

transfer_type: *USBTransferType* `= 2`

usage_type: *USBUsageType* `= 0`

class facedancer.**USBInterface**(*name: str = 'generic USB interface'*, *number: int = 0*, *alternate: int = 0*, *class_number: int = 0*, *subclass_number: int = 0*, *protocol_number: int = 0*, *interface_string: str = None*, *descriptors: ~typing.Dict[int, bytes] = <factory>*, *class_descriptor: bytes = None*, *endpoints: ~typing.Dict[int, ~facedancer.endpoint.USBEndpoint] = <factory>*, *parent: ~facedancer.descriptor.USBDescribable = None*)

Bases: *USBDescribable*, *AutoInstantiable*, *USBRequestHandler*

Class representing a USBDevice interface.

Fields:

number :
The interface's index. Zero indexed.

class_number, subclass_number, protocol_number :
The USB class adhered to on this interface; usually a USBDeviceClass constant.

interface_string :
A short, descriptive string used to identify the endpoint; or None if not provided.

DESCRIPTOR_TYPE_NUMBER `= 4`

add_endpoint(*endpoint:* USBEndpoint)

Adds the provided endpoint to the interface.

alternate: `int = 0`

class_descriptor: `bytes = None`

class_number: `int = 0`

descriptors: `Dict[int, bytes]`

endpoints: `Dict[int,` *USBEndpoint*`]`

classmethod from_binary_descriptor(*data*)

Generates an interface object from a descriptor.

get_descriptor() → bytes

Retrieves the given interface's interface descriptor, with subordinates.

get_device()

Returns the device associated with the given descriptor.

get_endpoint(*endpoint_number: int*, *direction:* USBDirection) → *USBEndpoint*

Attempts to find a subordinate endpoint matching the given number/direction.

Parameters

- **endpoint_number** – The endpoint number to search for.
- **direction** – The endpoint direction to be matched.

Returns

The matching endpoint; or None if no matching endpoint existed.

get_endpoints()

Returns an iterable over all endpoints in this interface.

get_identifier() → int

Returns a unique integer identifier for this object.

This is usually the index or address of the relevant USB object.

handle_buffer_empty(*endpoint:* USBEndpoint)

Handler called when a given endpoint first has an empty buffer.

Often, an empty buffer indicates an opportunity to queue data for sending ('prime an endpoint'), but doesn't necessarily mean that the host is planning on reading the data.

This function is called only once per buffer.

handle_data_received(*endpoint:* USBEndpoint, *data: bytes*)

Handler for receipt of non-control request data.

Typically, this method will delegate any data received to the appropriate configuration/interface/endpoint. If overridden, the overriding function will receive all data; and can delegate it by calling the *.handle_data_received* method on *self.configuration*.

Parameters

- **endpoint_number** – The endpoint number on which the data was received.
- **data** – The raw bytes received on the relevant endpoint.

handle_data_requested(*endpoint:* USBEndpoint)

Handler called when the host requests data on a non-control endpoint.

Typically, this method will delegate the request to the appropriate interface+endpoint. If overridden, the overriding function will receive all data.

Parameters

endpoint_number – The endpoint number on which the host requested data.

handle_get_descriptor_request = <ControlRequestHandler wrapping USBInterface.handle_get_descriptor_request at 0x7fae24d99700

handle_set_interface_request = <ControlRequestHandler wrapping USBInterface.handle_set_interface_request at 0x7fae24d992b0

has_endpoint(*endpoint_number: int*, *direction:* USBDirection) → *USBEndpoint*

Returns true iff we have matching subordinate endpoint.

Parameters

- **endpoint_number** – The endpoint number to search for.
- **direction** – The endpoint direction to be matched.

interface_string: str = None

name: str = 'generic USB interface'

number: int = 0

parent: *USBDescribable* = None

protocol_number: int = 0

subclass_number: int = 0

class facedancer.**USBRequestRecipient**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

Enumeration that describes each 'recipient' of a USB request field.

DEVICE = 0

ENDPOINT = 2

INTERFACE = 1

OTHER = 3

RESERVED = 4

classmethod from_integer(*value*)

Special factory that correctly handles reserved values.

classmethod from_request_type(*request_type_int*)

Helper method that extracts the type from a request_type integer.

class facedancer.**USBRequestType**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

Enumeration that describes each possible Type field for a USB request.

CLASS = 1

RESERVED = 3

STANDARD = 0

VENDOR = 2

classmethod **from_request_type**(*request_type_int*)

Helper method that extracts the type from a request_type integer.

class facedancer.**USBStandardRequests**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

CLEAR_FEATURE = 1

GET_CONFIGURATION = 8

GET_DESCRIPTOR = 6

GET_INTERFACE = 10

GET_STATUS = 0

SET_ADDRESS = 5

SET_CONFIGURATION = 9

SET_DESCRIPTOR = 7

SET_FEATURE = 3

SET_INTERFACE = 11

SYNCH_FRAME = 12

class facedancer.**USBSynchronizationType**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

ADAPTIVE = 2

ASYNC = 1

NONE = 0

SYNCHRONOUS = 3

class facedancer.**USBTransferType**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

BULK = 2

CONTROL = 0

INTERRUPT = 3

ISOCHRONOUS = 1

class facedancer.**USBUsageType**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

DATA = 0

FEEDBACK = 1

IMPLICIT_FEEDBACK = 2

facedancer.**class_request_handler**(***kwargs*)

Decorator; declares a class request handler. See control_request_handler() for usage.

facedancer.**standard_request_handler**(***kwargs*)

Decorator; declares a standard request handler. See control_request_handler() for usage.

facedancer.**to_any_endpoint**(*func*)

Decorator; refines a handler so it's only called on requests with an endpoint recipient.

facedancer.**to_any_interface**(*func*)

Decorator; refines a handler so it's only called on requests with an interface recipient.

facedancer.**to_device**(*func*)

Decorator; refines a handler so it's only called on requests with a device recipient.

facedancer.**to_other**(*func*)

Decorator; refines a handler so it's only called on requests with an Other (TM) recipient.

facedancer.**to_this_endpoint**(*func*)

Decorator; refines a handler so it's only called on requests targeting this endpoint.

facedancer.**to_this_interface**(*func*)

Decorator; refines a handler so it's only called on requests targeting this interface.

facedancer.**use_automatically**(*cls*)

Class decorator used to annotate FaceDancer inner classes. Implies @dataclass.

This decorator can be placed on inner classes that describe "subordinate" objects on USB devices. For example, a USBDevice can have several subordinate USBConfigurations; which select the various configurations for that class.

When placed on a subordinate class, this allows the parent class to automatically instantiate the relevant given class during its creation; automatically populating the subordinate properties of the relevant device.

For example, assume we have a FaceDancer class representing a custom USB device:

```
@dataclass
class ExampleDevice(USBDevice):
    product_string : str = "My Example Device"

    @use_automatically
    class DefaultConfiguration(USBConfiguration):
        number : int = 1
```

In this case, when an ExampleDevice is instantiated, the USBDevice code knows how to instantiate DefaultConfiguration, and will do so automatically.

Note that this decorator should _only_ be used for subordinate types; and expects that the decorated class has no explicitly-declared __init__ method. The __post_init__ mechanism of python dataclasses can be overridden to perform any needed initialization.

facedancer.**use_inner_classes_automatically**(*cls*)

Decorator that acts as if all inner classes were defined with *use_automatically*.

facedancer.**vendor_request_handler**(***kwargs*)

Decorator; declares a vendor request handler. See control_request_handler() for usage.

CHAPTER

SEVEN

HOW TO WRITE A NEW FACEDANCER BACKEND

Facedancer board backends can be found in the facedancer/backends/ directory.

To create a new backend, follow these steps:

7.1 1. Derive a new backend class

All Facedancer board backends inherit from the `FacedancerApp` and `FacedancerBackend` classes. Begin by deriving your new backend class from these base classes, as shown below:

```
from facedancer.core           import FacedancerApp
from facedancer.backends.base  import FacedancerBackend

class MydancerBackend(FacedancerApp, FacedancerBackend):

    app_name = "Mydancer"
```

7.2 2. Implement backend callback methods

Your new backend must implement the required callback methods defined in the `FacedancerBackend` class. These methods contain the functionality specific to your Facedancer board:

```
from typing    import List
from ..        import *


class FacedancerBackend:
    def __init__(self, device: USBDevice=None, verbose: int=0, quirks: List[str]=[]):
        """
        Initializes the backend.

        Args:
            device  :  The device that will act as our Facedancer.   (Optional)
            verbose : The verbosity level of the given application. (Optional)
            quirks  :  List of USB platform quirks.                  (Optional)
        """
        raise NotImplementedError

```

(continues on next page)

(continued from previous page)

```
    @classmethod
    def appropriate_for_environment(cls, backend_name: str) -> bool:
        """
        Determines if the current environment seems appropriate
        for using this backend.

        Args:
            backend_name : Backend name being requested. (Optional)
        """
        raise NotImplementedError


    def get_version(self):
        """
        Returns information about the active Facedancer version.
        """
        raise NotImplementedError


    def connect(self, usb_device: USBDevice, max_packet_size_ep0: int=64, device_speed:␣
↪DeviceSpeed=DeviceSpeed.FULL):
        """
        Prepares backend to connect to the target host and emulate
        a given device.

        Args:
            usb_device : The USBDevice object that represents the emulated device.
            max_packet_size_ep0 : Max packet size for control endpoint.
            device_speed : Requested usb speed for the Facedancer board.
        """
        raise NotImplementedError


    def disconnect(self):
        """ Disconnects Facedancer from the target host. """
        raise NotImplementedError


    def reset(self):
        """
        Triggers the Facedancer to handle its side of a bus reset.
        """
        raise NotImplementedError


    def set_address(self, address: int, defer: bool=False):
        """
        Sets the device address of the Facedancer. Usually only used during
        initial configuration.

        Args:
```

(continues on next page)

(continued from previous page)

```
            address : The address the Facedancer should assume.
            defer   : True iff the set_address request should wait for an active␣
↪transaction to
                      finish.
        """
        raise NotImplementedError


    def configured(self, configuration: USBConfiguration):
        """
        Callback that's issued when a USBDevice is configured, e.g. by the
        SET_CONFIGURATION request. Allows us to apply the new configuration.

        Args:
            configuration : The USBConfiguration object applied by the SET_CONFIG␣
↪request.
        """
        raise NotImplementedError


    def read_from_endpoint(self, endpoint_number: int) -> bytes:
        """
        Reads a block of data from the given endpoint.

        Args:
            endpoint_number : The number of the OUT endpoint on which data is to be rx'd.
        """
        raise NotImplementedError


    def send_on_endpoint(self, endpoint_number: int, data: bytes, blocking: bool=True):
        """
        Sends a collection of USB data on a given endpoint.

        Args:
            endpoint_number : The number of the IN endpoint on which data should be sent.
            data : The data to be sent.
            blocking : If true, this function should wait for the transfer to complete.
        """
        raise NotImplementedError


    def ack_status_stage(self, direction: USBDirection=USBDirection.OUT, endpoint_
↪number:int =0, blocking: bool=False):
        """
        Handles the status stage of a correctly completed control request,
        by priming the appropriate endpoint to handle the status phase.

        Args:
            direction : Determines if we're ACK'ing an IN or OUT vendor request.
                        (This should match the direction of the DATA stage.)
            endpoint_number : The endpoint number on which the control request
```

(continues on next page)

(continued from previous page)

```
                            occurred.
            blocking : True if we should wait for the ACK to be fully issued
                       before returning.
        """


    def stall_endpoint(self, endpoint_number:int, direction: USBDirection=USBDirection.
↪OUT):
        """
        Stalls the provided endpoint, as defined in the USB spec.

        Args:
            endpoint_number : The number of the endpoint to be stalled.
        """
        raise NotImplementedError


    def service_irqs(self):
        """
        Core routine of the Facedancer execution/event loop. Continuously monitors the
        Facedancer's execution status, and reacts as events occur.
        """
        raise NotImplementedError
```

7.3 3. Implement the backend event loop

Facedancer uses a polling approach to service events originating from the Facedancer board.

The actual events that need to be serviced will be specific to your Facedancer board but will generally include at least the following:

- Receiving a setup packet.
- Receiving data on an endpoint.
- Receiving NAK events (e.g. host requested data from an IN endpoint)

Facedancer will take care of scheduling execution of the `service_irqs()` callback but it is up to you to dispatch any events generated by your board to the corresponding methods of the Facedancer `USBDevice` object obtained in the `FacedancerBackend.connect()` callback.

That said, most backend implementations will follow a pattern similiar to the pseudo-code below:

```
class MydancerBackend(FacedancerApp, FacedancerBackend):

    ...

    def service_irqs(self):
        """
        Core routine of the Facedancer execution/event loop. Continuously monitors the
        Moondancer's execution status, and reacts as events occur.
        """
```

(continues on next page)

(continued from previous page)

```
        # obtain latest events and handle them
        for event in self.mydancer.get_events():
            match event:
                case USB_RECEIVE_SETUP:
                    self.usb_device.create_request(event.data)
                case USB_RECEIVE_PACKET:
                    self.usb_device.handle_data_available(event.endpoint_number, event.
↪data)
                case USB_EP_IN_NAK:
                    self.usb_device.handle_nak(event.endpoint_number)
```

Additionally, referencing the `service_irqs` methods of the other backend implementations can provide valuable insights into handling events specific to your implementation.

PYTHON MODULE INDEX

f

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z