nobodd 0.4 Documentation

Release 0.4

Dave Jones

Mar 07, 2024

CONTENTS

CHAPTER

ONE

INSTALLATION

nobodd is distributed in several formats. The following sections detail installation on a variety of platforms.

1.1 Ubuntu PPA

For Ubuntu, it may be simplest to install from the author's PPA[1] as follows:

```
$ sudo add-apt-repository ppa:waveform/nobodd
$ sudo apt install nobodd
```

If you wish to remove nobodd:

```
$ sudo apt remove nobodd
```

The deb-packaging includes a full man-page, and systemd service definitions.

1.2 Other Platforms

If your platform is *not* covered by one of the sections above, nobodd is available from PyPI and can therefore be installed with the Python setuptools "pip" tool:

```
$ pip install nobodd
```

On some platforms you may need to use a Python 3 specific alias of pip:

```
$ pip3 install nobodd
```

If you do not have either of these tools available, please install the Python setuptools[2] package first.

You can upgrade nobodd via pip:

```
$ pip install --upgrade nobodd
```

And removal can be performed as follows:

```
$ pip uninstall nobodd
```

[1] https://launchpad.net/~waveform/+archive/ubuntu/nobodd
[2] https://pypi.python.org/pypi/setuptools/

CHAPTER

TWO

TUTORIAL

nobodd is a confusingly named, but simple TFTP (Trivial File Transfer Protocol) server intended for net-booting Raspberry Pis directly from OS images without having to loop-back mount or otherwise re-write those images.

In order to get started you will need the following pre-requisites:

- A Raspberry Pi you wish to netboot. This tutorial will be assuming a Pi 4, but the Pi 2B, 3B, 3B+, 4B, and 5 all support netboot. However, all have subtly different means of configuring their netboot support, so in the interests of brevity this tutorial will only cover the method for the Pi 4.
- A micro-SD card. This is only required for the initial netboot configuration of the Pi 4, and for discovering the serial number of the board.
- A server that will serve the OS image to be netbooted. This can be another Raspberry Pi, but if you eventually wish to scale to several netbooting clients you probably want something with a lot more I/O bandwidth. We will assume this server is running Ubuntu 24.04, and you have root authority to install new packages.
- Ethernet networking connecting the two machines; netboot will *not* operate over WiFi.
- The addressing details of your ethernet network, specifically the network address and mask (e.g. 192.168.1.0/24).

2.1 Client Side

To configure your Pi 4 for netboot, use rpi-imager[3] to flash Ubuntu Server 24.04 64-bit to your micro-SD card. Boot your Pi 4 with the micro-SD card and wait for cloud-init[4] to finish the initial user configuration. Log in with the default user (username "ubuntu", password "ubuntu", unless you specified otherwise in rpi-imager), and follow the prompts to set a new password.

Run **`sudo rpi-eeprom-config --edit`**, and enter your password for "sudo". You will find yourself in an editor, with the Pi's boot configuration from the EEPROM, which will most likely look something like the following:

```
[all]
BOOT_UART=0
WAKE_ON_GPIO=1
ENABLE_SELF_UPDATE=1
BOOT_ORDER=0xf41
```

Note: Do not be concerned if several other values appear, or the ordering differs. Various versions of the Raspberry Pi boot EEPROM have had differing defaults for their configuration, and some later ones include a lot more values.

The value we are concerned with is `BOOT_ORDER` under the `[all]` section, which may be the only section in the file. This is a hexadecimal value (indicated by the "0x" prefix) in which each digit specifies another boot source in *reverse order*. The digits that may be specified include:

[3] https://www.raspberrypi.com/software/
[4] https://cloudinit.readthedocs.io/

#	Mode	Description
1	SD CARD	Boot from the SD card
2	NETWORK	Boot from TFTP over ethernet
4	USB-MSD	Boot from a USB MSD (mass storage device)
e	STOP	Stop the boot and display an error pattern
f	RESTART	Restart the boot from the first mode

A full listing[5] of valid digits can be found in the Raspberry Pi documentation. The current setting shown above is "0xf41". Remembering that this is in *reversed* order, we can interpret this as "try the SD card first (1), then try a USB mass storage device (4), then restart the sequence if neither worked (f)".

We'd like to try network booting first, so we need to add the value 2 to the end, giving us: "0xf412". Change the "BOOT_ORDER" value to this, save and exit the editor.

Warning: You may be tempted to remove values from the boot order to avoid delay (e.g. testing for the presence of an SD card). However, you are strongly advised to leave the value 1 (SD card booting) somewhere in your boot order to permit recovery from an SD card (or future re-configuration).

Upon exiting, the **`rpi-eeprom-config`** command should prompt you that you need to reboot in order to flash the new configuration onto the boot EEPROM. Enter **`sudo reboot`** to do so, and let the boot complete fully.

Once you are back at a login prompt, log back in with your username and password, and then run **`sudo rpi-eeprom-config`** once more to query the boot configuration and make sure your change has taken effect. It should output something like:

```
[all]
BOOT_UART=0
WAKE_ON_GPIO=1
ENABLE_SELF_UPDATE=1
BOOT_ORDER=0xf412
```

Finally, we need the serial number of your Raspberry Pi. This can be found with the following command.

```
$ grep ^Serial /proc/cpuinfo
Serial          : 10000000abcd1234
```

Note this number down somewhere safe as we'll need it for the server configuration later. The Raspberry Pi side of the configuration is now complete, and we can move on to configuring our netboot server.

2.2 Server Side

As mentioned in the pre-requisites, we will assume the server is running Ubuntu 24.04, and that you are logged in with a user that has root authority (via "sudo"). Firstly, install the packages which will provide our TFTP[6], NBD[7], and DHCP[8] proxy servers, along with some tooling to customize images.

```
$ sudo apt install nobodd-tftpd nobodd-tools nbd-server xz-utils dnsmasq
```

The first thing to do is configure *dnsmasq(8)* as a DHCP proxy server. Find the interface name of your server's primary ethernet interface (the one that will talk to the same network as the Raspberry Pi) within the output of the **`ip addr show up`** command. It will probably look something like "enp2s0f0".

[5] https://www.raspberrypi.com/documentation/computers/raspberry-pi.html#BOOT_ORDER
[6] https://en.wikipedia.org/wiki/Trivial_File_Transfer_Protocol
[7] https://en.wikipedia.org/wiki/Network_block_device
[8] https://en.wikipedia.org/wiki/Dynamic_Host_Configuration_Protocol

```
$ ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default␣
↪qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: enp2s0f0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group␣
↪default qlen 1000
    link/ether 0a:0b:0c:0d:0e:0f brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.4/16 brd 192.168.1.255 scope global enp2s0f0
       valid_lft forever preferred_lft forever
    inet6 fd00:abcd:1234::4/128 scope global noprefixroute
       valid_lft forever preferred_lft 53017sec
    inet6 fe80::beef:face:d00d:1234/64 scope link
        valid_lft forever preferred_lft forever
3: enp1s0f1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq master br0 state␣
↪UP group default qlen 1000
    link/ether 1a:0b:0c:0d:0e:0f brd ff:ff:ff:ff:ff:ff
4: br0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group␣
↪default qlen 1000
    link/ether 02:6c:fc:6f:56:5c brd ff:ff:ff:ff:ff:ff
    inet6 fe80::60d9:48ff:fee3:c955/64 scope link
       valid_lft forever preferred_lft forever
...
```

Add the following configuration lines to `/etc/dnsmasq.conf` adjusting the ethernet interface name, and the network mask on the highlighted lines to your particular setup.

```
# Only listen on the primary ethernet interface
interface=enp2s0f0
bind-interfaces

# Perform DHCP proxying on the network, and advertise our
# PXE-ish boot service
dhcp-range=192.168.1.255,proxy
pxe-service=0,"Raspberry Pi Boot"
```

Restart dnsmasq to ensure it's listening for DHCP connections (unfortunately reload is not sufficient in this case).

```
$ sudo systemctl restart dnsmasq.service
```

Next, we need to obtain an image to boot on our Raspberry Pi. We'll be using the Ubuntu 24.04 Server for Raspberry Pi image as this is configured for NBD boot out of the box. We will place this image under a `/srv/images` directory and unpack it so we can manipulate it.

```
$ sudo mkdir /srv/images
$ sudo chown ubuntu:ubuntu /srv/images
$ cd /srv/images
$ wget http://cdimage.ubuntu.com/releases/24.04/release/ubuntu-24.04-preinstalled-
↪server-arm64+raspi.img.xz
 ...
$ wget http://cdimage.ubuntu.com/releases/24.04/release/SHA256SUMS
 ...
$ sha256sum --check --ignore-missing SHA256SUMS
$ rm SHA256SUMS
$ unxz ubuntu-24.04-preinstalled-server-arm64+raspi.img.xz
```

We'll use the **nobodd-prep** command to adjust the image so that the kernel will try and find its root on our NBD server. At the same time, we'll have the utility generate the appropriate configurations for *nbd-server(1)* and **nobodd-tftpd**.

nobodd-prep needs to know several things in order to operate, but tries to use sensible defaults where it can:

- The filename of the image to customize; we'll simply provide this on the command line.
- The size we want to expand the image to; this will be size of the "disk" (or "SD card") that the Raspberry Pi sees. The default is 16GB, which is fine for our purposes here.
- The number of the boot partition within the image; the default is the first FAT partition, which is fine in this case.
- The name of the file containing the kernel command line on the boot partition; the default is `cmdline.txt` which is correct for the Ubuntu images.
- The number of the root partition within the image; the default is the first non-FAT partition, which is also fine here.
- The host-name of the server; the default is the output of **`hostname --fqdn`** but this can be specified manually with *nobodd-prep --nbd-host* (page 17).
- The name of the NBD share; the default is the stem of the image filename (the filename without its extensions) which in this case would be `ubuntu-24.04-preinstalled-server-arm64+raspi`. That's a bit of a mouthful so we'll override it with *nobodd-prep --nbd-name* (page 17).
- The serial number of the Raspberry Pi; there is no default for this, so we'll provide it with *nobodd-prep --serial* (page 18).
- The path to write the two configuration files we want to produce; we'll specify these manually with *nobodd-prep --tftpd-conf* (page 18) and *nobodd-prep --nbd-conf* (page 18)

Putting all this together we run,

```
$ nobodd-prep --nbd-name ubuntu-noble --serial 10000000abcd1234 \
> --tftpd-conf tftpd-noble.conf --nbd-conf nbd-noble.conf \
> ubuntu-24.04-preinstalled-server-arm64+raspi.img
```

Now we need to move the generated configuration files to their correct locations and ensure they're owned by root (so unprivileged users cannot modify them), ensure the modified image is owned by the "nbd" user (so the NBD service can read and write to it), and reload the configuration in the relevant services.

```
$ sudo chown nbd:nbd ubuntu-24.04-preinstalled-server-arm64+raspi.img
$ sudo chown root:root tftpd-noble.conf nbd-noble.conf
$ sudo mv tftpd-noble.conf /etc/nobodd/conf.d/
$ sudo mv nbd-noble.conf /etc/nbd-server/conf.d/
$ sudo systemctl reload nobodd-tftpd.service
$ sudo systemctl reload nbd-server.service
```

2.3 Testing and Troubleshooting

At this point your configuration should be ready to test. Ensure there is no SD card in the slot, and power it on. After a short delay you should see the "rainbow" boot screen appear. This will be followed by an uncharacteristically long delay on that screen. The reason is that your Pi is transferring the initramfs over TFTP which is not the most efficient protocol[9]. However, eventually you should be greeted by the typical Linux kernel log scrolling by, and reach a typical booted state the same as you would with a freshly flashed SD card.

If you hit any snags here, the following things are worth checking:

- Pay attention to any errors shown on the Pi's bootloader screen. In particular, you should be able to see the Pi obtaining an IP address via DHCP and various TFTP request attempts.
- Run `journalctl -f --unit nobodd-tftpd.service` on your server to follow the TFTP log output. Again, if things are working, you should be seeing several TFTP requests here. If you see nothing,

[9] absent certain extensions, which the Pi's bootloader doesn't implement.

double check the network mask is specified correctly in the *dnsmasq(8)* configuration, and that any firewall on your server is permitting inbound traffic to port 69 (the default TFTP port).

- You *will* see numerous "Early terminate" TFTP errors in the journal output. This is normal, and appears to be how the Pi's bootloader operates[10].

[10] at a guess it's attempting to determine the size of a file with the `tsize` extension, terminating the transfer, allocating RAM for the file, then starting the transfer again. While not *strictly* necessary, remember that the bootloader operates with limited resources and simplicity of operation is the order of the day.

CHAPTER

THREE

HOW TO GUIDES

The following guides cover specific, but commonly encountered, circumstances in operating a Raspberry Pi netboot server using NBD.

3.1 How to netboot Ubuntu 22.04

The Ubuntu 22.04 (jammy) images are not compatible with NBD boot out of the box as they lack the `nbd-client` package in their seed. However, you can modify the image to make it compatible.

3.1.1 On the Pi

Fire up rpi-imager[11] and flash Ubuntu 22.04.4 server onto an SD card, then boot that SD card on your Pi (the model does not matter provided it can boot the image).

> **Warning:** Do *not* be tempted to upgrade packages at this point. Specifically, the kernel package must *not* be upgraded yet.

Install the `linux-modules-extra-raspi` package for the currently running kernel version, and the `nbd-client` package.

```
$ sudo apt install linux-modules-extra-$(uname -r) nbd-client
```

On Ubuntu versions prior to 24.04, the `nbd` kernel module was moved out of the default `linux-modules-raspi` package for efficiency. We specifically need the version matching the running kernel version because installing this package will regenerate the initramfs (`initrd.img`). We'll be copying that regenerated file into the image we're going to netboot and it *must* match the kernel version in that image. This is why it was important not to upgrade any packages after the first boot.

We also need to install the NBD client package to add the `nbd-client` executable to the initramfs, along with some scripts to call it if the kernel command line specifies an NBD device as root:

We copy the regenerated `initrd.img` to the server, and shut down the Pi. Adjust the `ubuntu@server` reference below to fit your user on your server.

```
$ scp -q /boot/firmware/initrd.img ubuntu@server:
$ sudo poweroff
```

[11] https://www.raspberrypi.com/software/

3.1.2 On the Server

Download the same OS image to your server, verify its content, unpack it, and rename it to something more reasonable.

```
$ wget http://cdimage.ubuntu.com/releases/22.04.4/release/ubuntu-22.04.4-
↪preinstalled-server-arm64+raspi.img.xz
 ...
$ wget http://cdimage.ubuntu.com/releases/22.04.4/release/SHA256SUMS
 ...
$ sha256sum --check --ignore-missing SHA256SUMS
ubuntu-22.04.4-preinstalled-server-arm64+raspi.img.xz: OK
$ rm SHA256SUMS
$ mv ubuntu-22.04.4-preinstalled-server-arm64+raspi.img jammy.img
```

Next we need to create a cloud-init configuration which will perform the same steps we performed earlier on the first boot of our fresh image, namely to install `nbd-client` and `linux-modules-extra-raspi`, alongside the usual user configuration.

```
$ cat << EOF > user-data
#cloud-config

chpasswd:
  expire: true
  users:
  - name: ubuntu
    password: ubuntu
    type: text

ssh_pwauth: false

package_update: true
packages:
- nbd-client
- linux-modules-extra-raspi
EOF
```

See the cloud-init documentation[12], a this series of blog posts[13] for more ideas on what can be done with the `user-data` file.

3.1.3 Preparing the Image

When preparing our image with **`nobodd-prep`** we must remember to copy in our `user-data` and `initrd.img` files, overwriting the ones on the boot partition.

```
$ nobodd-prep --size 16GB --copy initrd.img --copy user-data jammy.img
```

At this point you should have a variant of the Ubuntu 22.04 image that is capable of being netbooted over NBD.

[12] https://cloudinit.readthedocs.io/
[13] https://waldorf.waveform.org.uk/tag/cloud-init.html

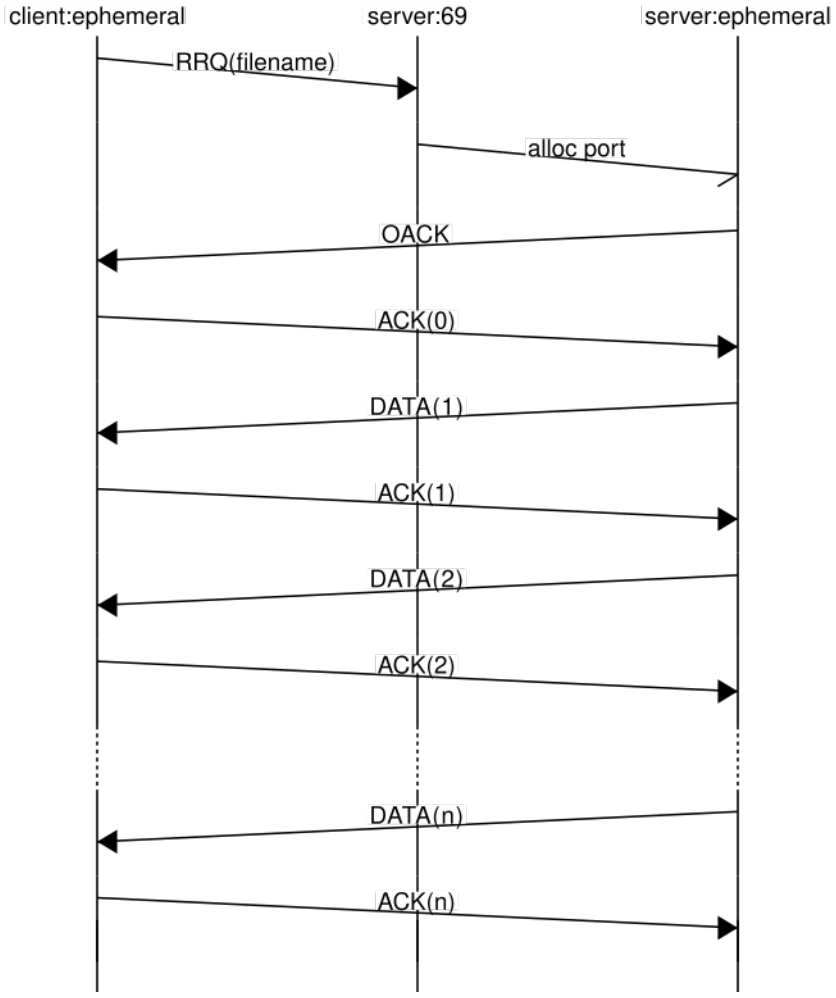
3.2 How to firewall your netboot server

If you wish to add a netfilter (or iptables) firewall to your server running nobodd and nbd-server, there are a few things to be aware of.

The NBD[14] protocol is quite trivial to firewall; the protocol uses TCP and listens on a single port: 10809. Hence, adding a rule that allows "NEW" inbound TCP connections on port 10809, and a rule to permit traffic on "ESTABLISHED" connections is generally sufficient (where "NEW" and "ESTABLISHED" have their typical meanings in netfilter's connection state tracking).

The TFTP[15] protocol is, theoretically at least, a little harder. The TFTP protocol uses UDP (i.e. it's connectionless) and though it starts on the privileged port[16] 69, this is only the case for the initial in-bound packet. All subsequent packets in a transfer take place on an ephemeral port on both the client *and the server*[17] .

Hence, a typical transfer looks like this:



Thankfully, because the server sends the initial response from its ephemeral port, and the client replies to that ephemeral port, it will also count as "ESTABLISHED" traffic in netfilter's parlance. Hence, all that's required to successfully firewall the TFTP side is to permit "NEW" inbound packets on port 69, and to permit "ESTABLISHED" UDP packets.

Putting this altogether, a typical `iptables(8)` sequence might look like this:

[14] https://en.wikipedia.org/wiki/Network_block_device
[15] https://en.wikipedia.org/wiki/Trivial_File_Transfer_Protocol
[16] https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers#Well-known_ports
[17] transfers are uniquely identified by the tuple of the client's ephemeral port, and the server's ephemeral port; this ensures a client may have multiple simultaneous transfers even in the case of a degenerate client that initiates multiple simultaneous transfers from a single port

```
$ sudo -i
[sudo] Password:
# iptables -A INPUT -p tcp -m state --state ESTABLISHED -j ACCEPT
# iptables -A INPUT -p tcp -m state --state NEW --dport 10809 -j ACCEPT
# iptables -A INPUT -p udp -m state --state ESTABLISHED -j ACCEPT
# iptables -A INPUT -p udp -m state --state NEW --dport 69 -j ACCEPT
```

CHAPTER

FOUR

EXPLANATIONS

The following chapter(s) contain explanations that may aid understanding of Raspberry Pi's netboot process in general.

4.1 Netboot on the Pi

In order to understand nobodd, it is useful to understand the netboot procedure on the Raspberry Pi in general. At a high level, it consists of three phases which we'll cover in the following sections.

4.1.1 DHCP

The first phase is quite simply a fairly typical *DHCP* phase, in which the bootloader attempts to obtain an IPv4 address from the local DHCP (Dynamic Host Configuration Protocol) server. On the Pi 4 (and later models), the address obtained can be seen on the boot diagnostics screen. Near the top the line starting with "net:" indicates the current network status. Initially this will read:

```
net: down ip: 0.0.0.0 sn: 0.0.0.0 gw: 0.0.0.0
```

Shortly before attempting netboot, this line should change to something like the following:

```
net: up ip: 192.168.1.137 sn: 255.255.255.0 gw: 192.168.1.1
```

This indicates that the Pi has obtained the address "192.168.1.137" on a class D subnet ("192.168.1.0/24" in CIDR[18] form), and knows the local network gateway is at "192.168.1.1".

The bootloader also inspects certain DHCP options to locate the TFTP[19] server for the next phase. Specifically:

- DHCP option 66 (TFTP server) can specify the address directly
- If DHCP option 43 (vendor options) specifies PXE string "Raspberry Pi Boot"[34] then option 54 (server identifier) will be used
- On the Pi 4 (and later), the EEPROM can override both of these with the TFTP_IP[20] option

With the network configured, and the TFTP server address obtained, we move onto the TFTP phase…

[18] https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing
[19] https://en.wikipedia.org/wiki/Trivial_File_Transfer_Protocol
[34] In early versions of the Raspberry Pi bootloader, the string needed to include three trailing spaces, i.e. `"Raspberry Pi Boot "`. Later versions of the bootloader perform a sub-string match.
[20] https://www.raspberrypi.com/documentation/computers/raspberry-pi.html#TFTP_IP

4.1.2 TFTP

Note: Most of the notes under this section are specific, in some way, to the netboot sequence on the Pi 4. While older and newer models may broadly follow the same sequence, there will be differences.

The bootloader's TFTP[21] client first attempts to locate the `start4.elf` file. By default, it looks for this in a directory named after the Pi's serial number. On the Pi 4 and later models, the EEPROM configuration can override this behaviour with the TFTP_PREFIX[22] option, but we will only cover the default behaviour here.

All subsequent files will be requested from within this serial number directory prefix[35]. Hence, when we say the bootloader requests `SERIAL/vmlinuz`, we mean it requests the file `vmlinuz` from within the virtual directory named after the Pi's serial number[36].

The attempt to retrieve `start4.elf` is immediately aborted when it is located, presumably because the intent is to determine the existence of the prefix directory, rather than the file itself. Next the bootloader attempts to read `SERIAL/config.txt`, which will configure the rest of the boot sequence.

Once `SERIAL/config.txt` has been retrieved, the bootloader parses it to discover the name of the tertiary bootloader to load[37], and requests `SERIAL/start.elf` or `SERIAL/start4.elf` (depending on the model) and the corresponding fix-up file (`SERIAL/fixup.dat` or `SERIAL/fixup4.dat` respectively).

The bootloader now executes the tertiary "start.elf" bootloader which requests `SERIAL/config.txt` again. This is re-parsed[38] and the name of the base device-tree, kernel, kernel command line, (optional) initramfs, and any (optional) device-tree overlays are determined. These are then requested over TFTP, placed in RAM, and finally the bootloader hands over control to the kernel.

TFTP Extensions

A brief aside on the subject of TFTP extensions (as defined in **RFC 2347**[23]). The basic TFTP protocol is extremely simple (as the acronym would suggest) and also rather inefficient, being limited to 512-byte blocks, in-order, synchronously (each block must be acknowledged before another can be sent), with no retry mechanism. Various extensions have been proposed to the protocol over the years, including those in **RFC 2347**[24], **RFC 2348**[25], **RFC 2349**[26], and **RFC 7440**[27].

The Pi bootloader implements *some* of these extensions. Specifically, it uses the "blocksize" extension (**RFC 2348**[28]) to negotiate a larger size of block to transfer, and the "tsize" extension (**RFC 2349**[29]) to attempt to determine the size of a transfer prior to it beginning.

However, its use of "tsize" is slightly unusual in that, when it finds the server supports it, it frequently starts a transfer with "tsize=0" (requesting the size of the file), but when the server responds with, for example, "tsize=1234" in the OACK packet (indicating the file to be transferred is 1234 bytes large), the bootloader then terminates the transfer.

In the case of the initial request for `start4.elf` (detailed above), this is understandable as a test for the existence of a directory, rather than an actual attempt to retrieve a file. However, in later requests the bootloader terminates the

[21] https://en.wikipedia.org/wiki/Trivial_File_Transfer_Protocol
[22] https://www.raspberrypi.com/documentation/computers/raspberry-pi.html#TFTP_IP
[35] If `start4.elf` is not found in the serial-number directory, the bootloader will attempt to lovate `start4.elf` with no directory prefix. If this succeeds, all subsequent requests will have no serial-number directory prefix.
[36] Some Pi serial numbers begin "10000000". This prefix is ignored for the purposes of constructing the serial-number directory prefix. For example, if the serial number is "10000000abcd1234", the `config.txt` file would be requested as `abcd1234/config.txt`.
[37] This does not happen on the Pi 5, which loads the tertiary bootloader from its (larger) EEPROM. On all prior models, the tertiary bootloader (start*.elf) loads from the boot medium, and the specific file loaded may be customized by `config.txt`.
[38] The tertiary bootloader operates on all `[sections]` in the `config.txt`. The secondary bootloader (`bootcode.bin`) only operates on some of these and doesn't comprehend the full syntax that the tertiary bootloader does (for instance, the secondary bootloader won't handle includes).
[23] https://datatracker.ietf.org/doc/html/rfc2347.html
[24] https://datatracker.ietf.org/doc/html/rfc2347.html
[25] https://datatracker.ietf.org/doc/html/rfc2348.html
[26] https://datatracker.ietf.org/doc/html/rfc2349.html
[27] https://datatracker.ietf.org/doc/html/rfc7440.html
[28] https://datatracker.ietf.org/doc/html/rfc2348.html
[29] https://datatracker.ietf.org/doc/html/rfc2349.html

transfer after the initial packet, *then immediately restarts it*. My best guess is that it allocates the RAM for the transfer after the termination, then restarts it (though why it does this is a bit of a mystery as it could allocate the space and continue the transfer, since the OACK packet doesn't contain any of the file data itself).

Sadly, the "windowsize" extension (**RFC 7440**[30]) is not yet implemented which means the Pi's netboot, up to the kernel, is quite slow compared to other methods.

4.1.3 Kernel

The kernel is now running with the configured command line, and (optionally) the address of an initial ramdisk (initramfs) as the root file-system. The initramfs is expected to contain the relevant kernel modules, and client binaries to talk to whatever network server will provide the root file-system.

Traditionally on the Raspberry Pi, this has meant NFS[31]. However, it may also be NBD[32] (as served by *nbd-server(1)*) or iSCSI[33] (as served by *iscsid(8)*). Typically, the `init` process loaded from the kernel's initramfs will dissect the kernel's command line to determine the location of the root file-system, and mount it using the appropriate utilities.

In the case of *nbd-server(1)* the following items in the kernel command line are crucial:

- `ip=dhcp` tells the kernel that it should request an IP address via DHCP (the Pi's bootloader cannot pass network state to the kernel, so this must be re-done)
- `nbdroot=HOST/SHARE` tells the kernel that it should open "SHARE" on the NBD server at HOST. This will form the block device `/dev/nbd0`
- `root=/dev/nbd0p2` tells the kernel that the root file-system is on the second partition of the block device

[30] https://datatracker.ietf.org/doc/html/rfc7440.html
[31] https://en.wikipedia.org/wiki/Network_File_System
[32] https://en.wikipedia.org/wiki/Network_block_device
[33] https://en.wikipedia.org/wiki/ISCSI

CHAPTER

FIVE

CLI REFERENCE

The following chapters document the command line utilities included in nobodd:

5.1 nobodd-prep

Customizes an OS image to prepare it for netbooting via TFTP. Specifically, this expands the image to a specified size (the assumption being the image is a copy of a minimally sized template image), then updates the kernel command line on the boot partition to point to an NBD server.

5.1.1 Synopsis

```
usage: nobodd-prep [-h] [--version] [-s SIZE] [--nbd-host HOST]
                   [--nbd-name NAME] [--cmdline NAME]
                   [--boot-partition NUM] [--root-partition NUM]
                   [-C PATH] [-R PATH] image
```

5.1.2 Options

image
: The target image to customize

-h, --help
: show the help message and exit

--version
: show program's version number and exit

-s SIZE, **--size** SIZE
: The size to expand the image to; default: 16GB

--nbd-host HOST
: The hostname of the nbd server to connect to for the root device; defaults to the local machine's FQDN

--nbd-name NAME
: The name of the nbd share to use as the root device; defaults to the stem of the *image* name

--cmdline NAME
: The name of the file containing the kernel command line on the boot partition; default: `cmdline.txt`

--boot-partition NUM
: Which partition is the boot partition within the image; default is the first FAT partition (identified by partition type) found in the image

--root-partition NUM

Which partition is the root partition within the image default is the first non-FAT partition (identified by partition type) found in the image

-C PATH, **--copy** PATH

Copy the specified file or directory into the boot partition. This may be given multiple times to specify multiple items to copy

-R PATH, **--remove** PATH

Delete the specified file or directory within the boot partition. This may be given multiple times to specify multiple items to delete

--serial HEX

Defines the serial number of the Raspberry Pi that will be served this image. When this option is given, a board configuration compatible with **nobodd-tftpd** may be output with *--tftpd-conf* (page 18)

--tftpd-conf FILE

If specified, write a board configuration compatible with **nobodd-tftpd** to the specified file; requires *--serial* (page 18) to be given. If "-" is given, output is written to stdout.

--nbd-conf FILE

If specified, write a share configuration compatible with *nbd-server(1)* to the specified file. If "-" is given, output is written to stdout.

5.1.3 Usage

Typically **nobodd-prep** is called with a base OS image. For example, if `ubuntu-24.04-server.img.xz` is the Ubuntu 24.04 Server for Raspberry image, we would decompress it (we can only work on uncompressed images), use the tool to expand it to a reasonable disk size (e.g. 16GB like an SD card), and customize the kernel command line to look for the rootfs on our NBD server:

```
$ ls -l ubuntu-24.04-server.img.xz
-rw-rw-r-- 1 dave dave 1189280360 Oct 12 00:44 ubuntu-24.04-server.img.xz
$ unxz ubuntu-24.04-server.img.xz
$ ls -l ubuntu-24.04-server.img
-rw-rw-r-- 1 dave dave 3727687680 Oct 12 00:44 ubuntu-24.04-server.img
$ fdisk -l ubuntu-24.04-server.img
Disk ubuntu-24.04-server.img: 3.47 GiB, 3727687680 bytes, 7280640 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x1634ec00

Device                    Boot   Start     End Sectors  Size Id Type
ubuntu-24.04-server.img1 *       2048 1050623 1048576  512M  c W95 FAT32 (LBA)
ubuntu-24.04-server.img2      1050624 7247259 6196636    3G 83 Linux
$ mkdir mnt
$ sudo mount -o loop,offset=$((2048*512)),sizelimit=$((1048576*512)) ubuntu-24.04-
↪server.img mnt/
[sudo] Password:
$ cat mnt/cmdline.txt
console=serial0,115200 multipath=off dwc_otg.lpm_enable=0 console=tty1␣
↪root=LABEL=writable rootfstype=ext4 rootwait fixrtc
$ sudo umount mnt/
$ nobodd-prep --size 16GB ubuntu-24.04-server.img
$ ls -l ubuntu-24.04-server.img --nbd-host myserver --nbd-name ubuntu
-rw-rw-r-- 1 dave dave 17179869184 Feb 27 13:11 ubuntu-24.04-server.img
$ sudo mount -o loop,offset=$((2048*512)),sizelimit=$((1048576*512)) ubuntu-24.04-
↪server.img mnt/
```

(continues on next page)

(continued from previous page)

```
[sudo] Password:
$ cat mnt/cmdline.txt
ip=dhcp nbdroot=myserver/ubuntu root=/dev/nbd0p2 console=serial0,115200␣
↪multipath=off dwc_otg.lpm_enable=0 console=tty1 rootfstype=ext4 rootwait fixrtc
$ sudo umount mnt/
```

Note, the only reason we are listing partitions and mounting the boot partition above is to demonstrate the change to the kernel command line in `cmdline.txt`. Ordinarily, usage of **nobodd-prep** is as simple as:

```
$ unxz ubuntu-24.04-server.img.xz
$ nobodd-prep --size 16GB ubuntu-24.04-server.img
```

Typically **nobodd-prep** will detect the boot and root partitions of the image automatically. The boot partition is defined as the first partition that has a FAT partition type[39] (on MBR-partitioned[40] images), or Basic Data[41] or EFI System[42] partition type (on GPT-partitioned[43] images), which contains a valid FAT file-system (the script tries to determine the FAT-type of the contained file-system, and only counts those partitions on which it can determine a valid FAT-type).

The root partition is the exact opposite; it is defined as the first partition that *doesn't* have a FAT partition type[44] (on MBR-partitioned[45] images), or Basic Data[46] or EFI System[47] partition type (on GPT-partitioned[48] images), which contains something *other than* a valid FAT file-system (again, the script tries to determine the FAT-type of the contained file-system, and only counts those partitions on which it *cannot* determine a valid FAT-type).

There may be images for which these simplistic definitions do not work. For example, images derived from a NOOBS/PINN[49] install may well have several boot partitions for different installed OS'. In this case the boot or root partition (or both) may be specified manually on the command line:

```
$ fdisk -l pinn-test.img
Disk pinn-test.img: 29.72 GiB, 31914983424 bytes, 62333952 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x2e779525

Device          Boot    Start      End  Sectors  Size Id Type
pinn-test.img1           8192   137215   129024   63M  e W95 FAT16 (LBA)
pinn-test.img2         137216 62333951 62196736 29.7G  5 Extended
pinn-test.img5         139264   204797    65534   32M 83 Linux
pinn-test.img6         204800   464895   260096  127M  c W95 FAT32 (LBA)
pinn-test.img7         466944  4661247  4194304    2G 83 Linux
pinn-test.img8        4669440  5193727   524288  256M 83 Linux
pinn-test.img9        5201920 34480125 29278206   14G 83 Linux
pinn-test.img10      34480128 34998271   518144  253M  c W95 FAT32 (LBA)
pinn-test.img11      35004416 62333951 27329536   13G 83 Linux
$ nobodd-prep --boot-partition 10 --root-partition 11 pinn-test.img
```

nobodd-prep also includes several facilities for customizing the boot partition beyond re-writing the kernel's `cmdline.txt`. Specifically, the *--remove* (page 18) and *--copy* (page 18) options.

The *--remove* (page 18) option can be given multiple times, and tells **nobodd-prep** to remove the specified files or directories from the boot partition. The *--copy* (page 18) option can also be given multiple times, and tells

[39] https://en.wikipedia.org/wiki/Partition_type
[40] https://en.wikipedia.org/wiki/Master_boot_record
[41] https://en.wikipedia.org/wiki/Microsoft_basic_data_partition
[42] https://en.wikipedia.org/wiki/EFI_system_partition
[43] https://en.wikipedia.org/wiki/GUID_Partition_Table
[44] https://en.wikipedia.org/wiki/Partition_type
[45] https://en.wikipedia.org/wiki/Master_boot_record
[46] https://en.wikipedia.org/wiki/Microsoft_basic_data_partition
[47] https://en.wikipedia.org/wiki/EFI_system_partition
[48] https://en.wikipedia.org/wiki/GUID_Partition_Table
[49] https://github.com/procount/pinn

nobodd-prep to copy the specified files or directories into the root of the boot partition. In both cases, directories that are specified are removed or copied recursively.

The *--copy* (page 18) option is particularly useful for overwriting the cloud-init[50] seeds on the boot partition of Ubuntu Server images, in case you want to provide an initial network configuration, user setup, or list of packages to install on first boot:

```
$ cat user-data
chpasswd:
  expire: true
  users:
  - name: ubuntu
    password: raspberry
    type: text

ssh_pwauth: false

package_update: true
package_upgrade: true
packages:
- avahi-daemon
$ nobodd-prep --copy user-data ubuntu-24.04-server.img
```

There is no need to *--remove* (page 18) files you wish to *--copy* (page 18); the latter option will overwrite where necessary. The exception to this is copying directories; if you are copying a directory that already exists in the boot partition, the new content will be merged with the existing content. Files under the directory that share a name will be overwritten, files that do not will be left in place. If you wish to replace the directory wholesale, specify it with *--remove* (page 18) as well.

The ordering of options on the command line does *not* affect the order of operations in the utility. The order of operations in **nobodd-prep** is strictly as follows:

1. Detect partitions, if necessary
2. Re-size the image, if necessary
3. Remove all items on the boot partition specified by *--remove* (page 18)
4. Copy all items specified by *--copy* (page 18) into the boot partition
5. Re-write the `root=` option in the `cmdline.txt` file

This ordering is deliberate, firstly to ensure directories can be replaced (as noted above), and secondly to ensure `cmdline.txt` can be customized by *--copy* (page 18) prior to the customization performed by the utility.

5.1.4 See Also

nobodd-tftpd (page 21), *nbd-server(1)*

5.1.5 Bugs

Please report bugs at: https://github.com/waveform80/nobodd/issues

[50] https://cloudinit.readthedocs.io/en/latest/

5.2 nobodd-tftpd

A read-only TFTP server capable of reading FAT boot partitions from within image files or devices. Intended to be paired with a block-device service (e.g. NBD) for netbooting Raspberry Pis.

5.2.1 Synopsis

```
usage: nobodd-tftpd [-h] [--version] [--listen ADDR] [--port PORT]
                    [--board SERIAL,FILENAME[,PART[,IP]]]
```

5.2.2 Options

-h, --help
show the help message and exit

--version
show program's version number and exit

--board SERIAL,FILENAME[,PART[,IP]]
can be specified multiple times to define boards which are to be served boot images over TFTP; if PART is omitted the default is 1; if IP is omitted the IP address will not be checked

--listen ADDR
the address on which to listen for connections (default: "::" for all addresses)

--port PORT
the port on which to listen for connections (default: "tftp" which is port 69)

5.2.3 Configuration

nobodd-tftpd can be configured via the command line, or from several configuration files. These are structured as INI-style files with bracketed `[sections]` containing `key=value` lines, and optionally #-prefixed comments. The configuration files which are read, and the order they are consulted is as follows:

1. `/etc/nobodd/nobodd.conf`
2. `/usr/local/etc/nobodd/nobodd.conf`
3. `$XDG_CONFIG_HOME/nobodd/nobodd.conf` (where `$XDG_CONFIG_HOME` defaults to `~/.config` if unset)

Later files override settings from files earlier in this order.

The configuration file may contain a `[tftp]` section which may contain the following values:

listen
This is equivalent to the *--listen* (page 21) parameter and specifies the address(es) on which the server will listen for incoming TFTP connections.

port
This is equivalent to the *--port* (page 21) parameter and specifies the UDP port on which the server will listen for incoming TFTP connections. Please note that only the *initial* TFTP packet will arrive on this port. Each "connection" is allocated its own ephemeral port[51] on the server and all subsequent packets will use this ephemeral port.

[51] https://en.wikipedia.org/wiki/Ephemeral_port

includedir
: If this is specified, it provides the name of a directory which will be scanned for files matching the pattern `*.conf`. Any files found matching will be read as additional configuration files, in sorted filename order.

For example:

```
[tftp]
listen = 192.168.0.0/16
port = tftp
includedir = /etc/nobodd/conf.d
```

For each image the TFTP server is expected to serve to a Raspberry Pi, a `[board:SERIAL]` section should be defined. Here, "SERIAL" should be replaced by the serial number of the Raspberry Pi. The serial number can be found in the output of `cat /proc/cpuinfo` at runtime. For example:

```
$ grep ^Serial /proc/cpuinfo
Serial          : 100000001234abcd
```

If the serial number starts with 10000000 (as in the example above), exclude the initial one and all leading zeros. So the above Pi has a serial number of 1234abcd (in hexadecimal). Within the section the following values are valid:

image
: Specifies the full path to the operating system image to serve to the specified Pi, presumably prepared with **nobodd-prep**.

partition
: Optionally specifies the number of the boot partition. If this is not specified it defaults to 1.

ip
: Optionally limits serving any files from this image unless the IP address of the client matches. If this is not specified, any IP address may retrieve files from this share.

For example:

```
[board:1234abcd]
image = /srv/images/ubuntu-24.04-server.img
partition = 1
ip = 192.168.0.5
```

In practice, what this means is that requests from a client with the IP address "192.168.0.5", for files under the path "1234abcd/", will be served from the FAT file-system on partition 1 of the image stored at `/srv/images/ubuntu-24.04-server.img`.

Such definitions can be produced by **nobodd-prep** when it is provided with the *nobodd-prep --serial* (page 18) option.

Boards may also be defined on the command-line with the *--board* (page 21) option. These definitions will augment (and override, where the serial number is identical) those definitions provided by the configuration files.

5.2.4 Systemd/Inetd Usage

The server may inherit its listening socket from a managing process. In the case of *inetd(8)* where the listening socket is traditionally passed as stdin (fd 0), pass "stdin" as the value of *--listen* (page 21) (or the `listen` option within the `[tftp]` section of the configuration file).

In the case of *systemd(1)*, where the listening socket(s) are passed via the environment, specify "systemd" as the value of *--listen* (page 21) (or the `listen` option within the `[tftp]` section of the configuration file) and the service will expect to find a single socket passed in `LISTEN_FDS`. This will happen implicitly if the service is declared as socket-activated. However, the service must *not* use `Accept=yes` as the TFTP protocol is connectionless. The example units provided in the source code demonstrate using socket-activation with the server.

In both cases, the service manager sets the port that the service will listen on, so the *--port* (page 21) option (and the `port` option in the `[tftp]` section of the configuration file) is silently ignored.

5.2.5 See Also

nobodd-prep (page 17), *nbd-server(1)*

5.2.6 Bugs

Please report bugs at: https://github.com/waveform80/nobodd/issues

CHAPTER

SIX

API REFERENCE

In additional to being a service, nobodd can also be used as an API from Python to access disk images, determining their partitioning style, enumerating the available partitions, and manipulating FAT file-systems (either from within a disk image, or just standalone). It can also be used as the basis of a generic TFTP service.

The following sections list the modules by their topic.

6.1 Disk Images

The *nobodd.disk.DiskImage* (page 26) class is the primary entry-point for dealing with disk images.

6.1.1 nobodd.disk

The *nobodd.disk* (page 25) module contains the *DiskImage* (page 26) class which is the primary entry point for handling disk images. Constructed with a filename (or file-like object which provides a valid `fileno()`[52] method), the class will attempt to determine if MBR[53] or GPT[54] style partitioning is in use. The *DiskImage.partitions* (page 26) attribute can then be queried to enumerate, or access the data of, individual partitions:

```
>>> from nobodd.disk import DiskImage
>>> img = DiskImage('gpt_disk.img')
>>> img
<DiskImage file=<_io.BufferedReader name='gpt_disk.img'> style='gpt' signature=UUID('733b49a8-6918-4e44-8d3d-47ed9b481335')>
>>> img.style
'gpt'
>>> len(img.partitions)
4
>>> img.partitions
DiskPartitionsGPT({
1: <DiskPartition size=8388608 label='big-part' type=UUID('ebd0a0a2-b9e5-4433-87c0-68b6b72699c7')>,
2: <DiskPartition size=204800 label='little-part1' type=UUID('ebd0a0a2-b9e5-4433-87c0-68b6b72699c7')>,
5: <DiskPartition size=4194304 label='medium-part' type=UUID('ebd0a0a2-b9e5-4433-87c0-68b6b72699c7')>,
6: <DiskPartition size=204800 label='little-part2' type=UUID('ebd0a0a2-b9e5-4433-87c0-68b6b72699c7')>,
})
```

Note that partitions are numbered from 1 and that, especially in the case of MBR[55], partition numbers may not be contiguous: primary partitions are numbered 1 through 4, but logical partitions may only exist in one primary partition, and are numbered from 5. Hence it is entirely valid to have partitions 1, 5, and 6:

[52] https://docs.python.org/3.12/library/io.html#io.IOBase.fileno
[53] https://en.wikipedia.org/wiki/Master_boot_record
[54] https://en.wikipedia.org/wiki/GUID_Partition_Table
[55] https://en.wikipedia.org/wiki/Master_boot_record

```
>>> from nobodd.disk import DiskImage
>>> img = DiskImage('test-ebr.img')
>>> img.style
'mbr'
>>> len(img.partitions)
3
>>> list(img.partitions.keys())
[1, 5, 6]
>>> img.partitions[1]
<DiskPartition size=536870912 label='Partition 1' type=12>
>>> img.partitions[5]
<DiskPartition size=536870912 label='Partition 5' type=131>
>>> img.partitions[6]
<DiskPartition size=1070596096 label='Partition 6' type=131>
```

GPT[56] partition tables may also have non-contiguous numbering, although this is less common in practice. The *DiskPartition.data* (page 27) attribute can be used to access the content of the partition as a buffer object (see memoryview[57]).

DiskImage

class nobodd.disk.**DiskImage**(*filename_or_obj*, *sector_size=512*, *access=1*)

Represents a disk image, specified by *filename_or_obj* which must be a str[58] or Path[59] naming the file, or a file-like object.

If a file-like object is provided, it *must* have a fileno[60] method which returns a valid file-descriptor number (the class uses mmap[61] internally which requires a "real" file).

The disk image is expected to be partitioned with either an MBR[62] partition table or a GPT[63]. The partitions within the image can be enumerated with the *partitions* (page 26) attribute. The instance can (and should) be used as a context manager; exiting the context will call the *close()* (page 26) method implicitly.

If specified, *sector_size* is the size of sectors (in bytes) within the disk image. This defaults to 512 bytes, and should almost always be left alone. The *access* parameter controls the access used when constructing the memory mapping. This defaults to mmap.ACCESS_READ for read-only access. If you wish to write to file-systems within the disk image, change this to mmap.ACCESS_WRITE. You may also use mmap.ACCESS_COPY for read-write mappings that don't actually affect the underlying disk image.

Note: Please note that this library provides no means to re-partition disk images, just the ability to re-write files within FAT partitions.

---

close()

Destroys the memory mapping used on the file provided. If the file was opened by this class, it will also be closed. This method is idempotent and is implicitly called when the instance is used as a context manager.

Note: All mappings derived from this one *must* be closed before calling this method. By far the easiest means of arranging this is to consistently use context managers with all instances derived from this.

---

property partitions

Provides access to the partitions in the image as a Mapping[64] of partition number to *DiskPartition* (page 27) instances.

[56] https://en.wikipedia.org/wiki/GUID_Partition_Table
[57] https://docs.python.org/3.12/library/stdtypes.html#memoryview

> **Warning:** Disk partition numbers start from 1 and need not be contiguous, or ordered.
>
> For example, it is perfectly valid to have partition 1 occur later on disk than partition 2, for partition 3 to be undefined, and partition 4 to be defined between partition 1 and 2. The partition number is essentially little more than an arbitrary key.
>
> In the case of MBR partition tables, it is particularly common to have missing partition numbers as the primary layout only permits 4 partitions. Hence, the "extended partitions" scheme numbers partitions from 5. However, if not all primary partitions are defined, there will be a "jump" from, say, partition 2 to partition 5.

property signature

The identifying signature of the disk. In the case of a GPT partitioned disk, this is a `UUID`[65]. In the case of MBR, this is a 32-bit integer number.

property style

The style of partition table in use on the disk image. Will be one of the strings, 'gpt' or 'mbr'.

DiskPartition

class nobodd.disk.**DiskPartition**(*mem*, *label*, *type*)

Represents an individual disk partition within a *DiskImage* (page 26).

Instances of this class are returned as the values of the mapping provided by *DiskImage.partitions* (page 26). Instances can (and should) be used as a context manager to implicitly close references upon exiting the context.

close()

Release the internal `memoryview`[66] reference. This method is idempotent and is implicitly called when the instance is used as a context manager.

property data

Returns a buffer (specifically, a `memoryview`[67]) covering the contents of the partition in the owning *DiskImage* (page 26).

property label

The label of the partition. GPT[68] partitions may have a 36 character unicode label. MBR[69] partitions do not have a label, so the string "Partition {num}" will be used instead (where *{num}* is the partition number).

property type

The type of the partition. For GPT[70] partitions, this will be a `uuid.UUID`[71] instance. For MBR[72] partitions, this will be an `int`[73].

[58] https://docs.python.org/3.12/library/stdtypes.html#str
[59] https://docs.python.org/3.12/library/pathlib.html#pathlib.Path
[60] https://docs.python.org/3.12/library/io.html#io.IOBase.fileno
[61] https://docs.python.org/3.12/library/mmap.html#mmap.mmap
[62] https://en.wikipedia.org/wiki/Master_boot_record
[63] https://en.wikipedia.org/wiki/GUID_Partition_Table
[64] https://docs.python.org/3.12/library/collections.abc.html#collections.abc.Mapping
[65] https://docs.python.org/3.12/library/uuid.html#uuid.UUID
[66] https://docs.python.org/3.12/library/stdtypes.html#memoryview
[67] https://docs.python.org/3.12/library/stdtypes.html#memoryview
[68] https://en.wikipedia.org/wiki/GUID_Partition_Table
[69] https://en.wikipedia.org/wiki/Master_boot_record
[70] https://en.wikipedia.org/wiki/GUID_Partition_Table
[71] https://docs.python.org/3.12/library/uuid.html#uuid.UUID
[72] https://en.wikipedia.org/wiki/Master_boot_record
[73] https://docs.python.org/3.12/library/functions.html#int

Internal Classes

You should not need to use these classes directly; they will be instantiated automatically when querying the *DiskImage.partitions* (page 26) attribute according to the detected table format.

class nobodd.disk.**DiskPartitionsGPT**(*mem*, *sector_size=512*)

Provides a Mapping[74] from partition number to *DiskPartition* (page 27) instances for a GPT[75].

mem is the buffer covering the whole disk image. *sector_size* specifies the sector size of the disk image, which should almost always be left at the default of 512 bytes.

class nobodd.disk.**DiskPartitionsMBR**(*mem*, *sector_size=512*)

Provides a Mapping[76] from partition number to *DiskPartition* (page 27) instances for a MBR[77].

mem is the buffer covering the whole disk image. *sector_size* specifies the sector size of the disk image, which should almost always be left at the default of 512 bytes.

6.1.2 nobodd.gpt

Defines the data structures used by GUID Partition Tables[78]. You should never need these directly; use the *nobodd.disk.DiskImage* (page 26) class instead.

Data Structures

class nobodd.gpt.**GPTHeader**(*signature*, *revision*, *header_size*, *header_crc32*, *current_lba*, *backup_lba*, *first_usable_lba*, *last_usable_lba*, *disk_guid*, *part_table_lba*, *part_table_size*, *part_entry_size*, *part_table_crc32*)

A namedtuple()[79] representing the fields of the GPT header[80].

classmethod from_buffer(*buf*, *offset=0*)

Construct a *GPTHeader* (page 28) from the specified *offset* (which defaults to 0) in the buffer protocol object, *buf*.

classmethod from_bytes(*s*)

Construct a *GPTHeader* (page 28) from the byte-string *s*.

class nobodd.gpt.**GPTPartition**(*type_guid*, *part_guid*, *first_lba*, *last_lba*, *flags*, *part_label*)

A namedtuple()[81] representing the fields of a GPT entry[82].

classmethod from_buffer(*buf*, *offset=0*)

Construct a *GPTPartition* (page 28) from the specified *offset* (which defaults to 0) in the buffer protocol object, *buf*.

classmethod from_bytes(*s*)

Construct a *GPTPartition* (page 28) from the byte-string *s*.

[74] https://docs.python.org/3.12/library/collections.abc.html#collections.abc.Mapping
[75] https://en.wikipedia.org/wiki/GUID_Partition_Table
[76] https://docs.python.org/3.12/library/collections.abc.html#collections.abc.Mapping
[77] https://en.wikipedia.org/wiki/Master_boot_record
[78] https://en.wikipedia.org/wiki/GUID_Partition_Table
[79] https://docs.python.org/3.12/library/collections.html#collections.namedtuple
[80] https://en.wikipedia.org/wiki/GUID_Partition_Table#Partition_table_header_(LBA_1)
[81] https://docs.python.org/3.12/library/collections.html#collections.namedtuple
[82] https://en.wikipedia.org/wiki/GUID_Partition_Table#Partition_entries_(LBA_2%E2%80%9333)

6.1.3 nobodd.mbr

Defines the data structures used by the Master Boot Record[83] (MBR) partitioning style. You should never need these directly; use the *nobodd.disk.DiskImage* (page 26) class instead.

Data Structures

class nobodd.mbr.**MBRHeader**(*zero*, *physical_drive*, *seconds*, *minutes*, *hours*, *disk_sig*, *copy_protect*, *partition_1*, *partition_2*, *partition_3*, *partition_4*, *boot_sig*)

A namedtuple()[84] representing the fields of the MBR header[85].

classmethod from_buffer(*buf*, *offset=0*)

Construct a *MBRHeader* (page 29) from the specified *offset* (which defaults to 0) in the buffer protocol object, *buf*.

classmethod from_bytes(*s*)

Construct a *MBRHeader* (page 29) from the byte-string *s*.

property partitions

Returns a sequence of the partitions defined by the header. This is always 4 elements long, and not all elements are guaranteed to be valid, or in order on the disk.

class nobodd.mbr.**MBRPartition**(*status*, *first_chs*, *part_type*, *last_chs*, *first_lba*, *part_size*)

A namedtuple()[86] representing the fields of an MBR partition entry[87].

classmethod from_buffer(*buf*, *offset=0*)

Construct a *MBRPartition* (page 29) from the specified *offset* (which defaults to 0) in the buffer protocol object, *buf*.

classmethod from_bytes(*s*)

Construct a *MBRPartition* (page 29) from the byte-string *s*.

6.2 FAT Filesystem

The *nobodd.fs.FatFileSystem* (page 30) class is the primary entry-point for handling FAT file-systems.

6.2.1 nobodd.fs

The *nobodd.fs* (page 29) module contains the *FatFileSystem* (page 30) class which is the primary entry point for reading FAT file-systems. Constructed with a buffer object representing a memory mapping of the file-system, the class will determine whether the format is FAT12, FAT16, or FAT32. The *root* (page 31) attribute provides a Path-like object representing the root directory of the file-system.

```
>>> from nobodd.disk import DiskImage
>>> from nobodd.fs import FatFileSystem
>>> img = DiskImage('test-gpt.img')
>>> fs = FatFileSystem(img.partitions[1].data)
>>> fs.fat_type
'fat16'
>>> fs.root
FatPath(<FatFileSystem label='TEST' fat_type='fat16'>, '/')
```

[83] https://en.wikipedia.org/wiki/Master_boot_record
[84] https://docs.python.org/3.12/library/collections.html#collections.namedtuple
[85] https://en.wikipedia.org/wiki/Master_boot_record#Sector_layout
[86] https://docs.python.org/3.12/library/collections.html#collections.namedtuple
[87] https://en.wikipedia.org/wiki/Master_boot_record#Partition_table_entries

Warning: At the time of writing, the implementation is strictly *not thread-safe*. Attempting to write to the file-system from multiple threads (whether in separate instances or not) is likely to result in corruption. Attempting to write to the file-system from one thread, while reading from another will result in undefined behaviour including incorrect reads.

Warning: The implementation will *not* handle certain "obscure" extensions to FAT, such as sub-directory style roots on FAT-12/16. It will attempt to warn about these and abort if they are found.

FatFileSystem

class nobodd.fs.**FatFileSystem**(*mem*, *atime=False*, *encoding='iso-8859-1'*)

Represents a FAT[88] file-system, contained at the start of the buffer object *mem*.

This class supports the FAT-12, FAT-16, and FAT-32 formats, and will automatically determine which to use from the headers found at the start of *mem*. The type in use may be queried from *fat_type* (page 38). Of primary use is the *root* (page 31) attribute which provides a *FatPath* (page 42) instance representing the root directory of the file-system.

Instances can (and should) be used as a context manager; exiting the context will call the *close()* (page 30) method implicitly. If certain header bits are set, *DamagedFileSystem* (page 33) and *DirtyFileSystem* (page 33) warnings may be generated upon opening.

If *atime* is False[89], the default, then accesses to files will *not* update the atime field in file meta-data (when the underlying *mem* mapping is writable). Finally, *encoding* specifies the character set used for decoding and encoding DOS short filenames.

close()

Releases the memory references derived from the buffer the instance was constructed with. This method is idempotent.

open_dir(*cluster*)

Opens the sub-directory in the specified *cluster*, returning a *FatDirectory* (page 35) instance representing it.

Warning: This method is intended for internal use by the *FatPath* (page 42) class.

open_entry(*index*, *entry*, *mode='rb'*)

Opens the specified *entry*, which must be a *DirectoryEntry* (page 40) instance, which must be a member of *index*, an instance of *FatDirectory* (page 35). Returns a *FatFile* (page 32) instance associated with the specified *entry*. This permits writes to the file to be properly recorded in the corresponding directory entry.

Warning: This method is intended for internal use by the *FatPath* (page 42) class.

open_file(*cluster*, *mode='rb'*)

Opens the file at the specified *cluster*, returning a *FatFile* (page 32) instance representing it with the specified *mode*. Note that the *FatFile* (page 32) instance returned by this method has no directory entry associated with it.

Warning: This method is intended for internal use by the *FatPath* (page 42) class, specifically for "files" underlying the sub-directory structure which do not have an associated size (other than that dictated by their FAT chain of clusters).

property atime

If the underlying mapping is writable, then atime (last access time) will be updated upon reading the content of files, when this property is `True`[90] (the default is `False`[91]).

property clusters

A *FatClusters* (page 35) sequence representing the clusters containing the data stored in the file-system.

> **Warning:** This attribute is intended for internal use by the *FatFile* (page 32) class, but may be useful for low-level exploration or manipulation of FAT file-systems.

property fat

A *FatTable* (page 33) sequence representing the FAT table itself.

> **Warning:** This attribute is intended for internal use by the *FatFile* (page 32) class, but may be useful for low-level exploration or manipulation of FAT file-systems.

property fat_type

Returns a `str`[92] indicating the type of FAT[93] file-system present. Returns one of "fat12", "fat16", or "fat32".

property label

Returns the label from the header of the file-system. This is an ASCII string up to 11 characters long.

property readonly

Returns `True`[94] if the underlying buffer is read-only.

property root

Returns a *FatPath* (page 42) instance (a `Path`[95]-like object) representing the root directory of the FAT file-system. For example:

```
from nobodd.disk import DiskImage
from nobodd.fs import FatFileSystem

with DiskImage('test.img') as img:
    with FatFileSystem(img.partitions[1].data) as fs:
        print('ls /')
        for p in fs.root.iterdir():
            print(p.name)
```

> **Note:** This is intended to be the primary entry-point for querying and manipulating the file-system at the high level. Only use the *fat* (page 31) and *clusters* (page 31) attributes, and the various "open" methods if you want to explore or manipulate the file-system at a low level.

property sfn_encoding

The encoding used for short (8.3) filenames. This defaults to "iso-8859-1" but unfortunately there's no way of determining the correct codepage for these.

[88] https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system
[89] https://docs.python.org/3.12/library/constants.html#False
[90] https://docs.python.org/3.12/library/constants.html#True
[91] https://docs.python.org/3.12/library/constants.html#False
[92] https://docs.python.org/3.12/library/stdtypes.html#str
[93] https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system
[94] https://docs.python.org/3.12/library/constants.html#True
[95] https://docs.python.org/3.12/library/pathlib.html#pathlib.Path

FatFile

class nobodd.fs.**FatFile**(*fs*, *start*, *mode='rb'*, *index=None*, *entry=None*)

Represents an open file from a *FatFileSystem* (page 30).

You should never need to construct this instance directly. Instead it (or wrapped variants of it) is returned by the *open()* (page 44) method of *FatPath* (page 42) instances. For example:

```
from nobodd.disk import DiskImage
from nobodd.fs import FatFileSystem

with DiskImage('test.img') as img:
    with FatFileSystem(img.partitions[1].data) as fs:
        path = fs.root / 'bar.txt'
        with path.open('r', encoding='utf-8') as f:
            print(f.read())
```

Instances can (and should) be used as context managers to implicitly close references upon exiting the context. Instances are readable and seekable, and writable, depending on their opening mode and the nature of the underlying *FatFileSystem* (page 30).

As a derivative of io.RawIOBase[96], all the usual I/O methods should be available.

close()

Flush and close the IO object.

This method has no effect if the file is already closed.

classmethod from_cluster(*fs*, *start*, *mode='rb'*)

Construct a *FatFile* (page 32) from a *FatFileSystem* (page 30), *fs*, and a *start* cluster. The optional *mode* is equivalent to the built-in open()[97] function.

Files constructed via this method do not have an associated directory entry. As a result, their size is assumed to be the full size of their cluster chain. This is typically used for the "file" backing a *FatSubDirectory* (page 37).

> **Warning:** This method is intended for internal use by the *FatPath* (page 42) class.

classmethod from_entry(*fs*, *index*, *entry*, *mode='rb'*)

Construct a *FatFile* (page 32) from a *FatFileSystem* (page 30), *fs*, a *FatDirectory* (page 35), *index*, and a *DirectoryEntry* (page 40), *entry*. The optional *mode* is equivalent to the built-in open()[98] function.

Files constructed via this method have an associated directory entry which will be updated if/when reads or writes occur (updating atime, mtime, and size fields).

> **Warning:** This method is intended for internal use by the *FatPath* (page 42) class.

readable()

Return whether object was opened for reading.

If False, read() will raise OSError.

readall()

Read until EOF, using multiple read() call.

seek(*pos*, *whence=0*)

Change stream position.

Change the stream position to the given byte offset. The offset is interpreted relative to the position indicated by whence. Values for whence are:

- 0 – start of stream (the default); offset should be zero or positive
- 1 – current stream position; offset may be negative
- 2 – end of stream; offset is usually negative

Return the new absolute position.

seekable()

Return whether object supports random access.

If False, seek(), tell() and truncate() will raise OSError. This method may need to do a test seek().

truncate(*size=None*)

Truncate file to size bytes.

File pointer is left unchanged. Size defaults to the current IO position as reported by tell(). Returns the new size.

writable()

Return whether object was opened for writing.

If False, write() will raise OSError.

Exceptions and Warnings

exception nobodd.fs.**FatWarning**

Base class for warnings issued by *FatFileSystem* (page 30).

exception nobodd.fs.**DirtyFileSystem**

Raised when opening a FAT file-system that has the "dirty" flag set in the second entry of the FAT.

exception nobodd.fs.**DamagedFileSystem**

Raised when opening a FAT file-system that has the I/O errors flag set in the second entry of the FAT.

exception nobodd.fs.**OrphanedLongFilename**

Raised when a *LongFilenameEntry* (page 40) is found with a mismatched checksum, terminal flag, out of order index, etc. This usually indicates an orphaned entry as the result of a non-LFN aware file-system driver manipulating a directory.

exception nobodd.fs.**BadLongFilename**

Raised when a *LongFilenameEntry* (page 40) is unambiguously corrupted, e.g. including a non-zero cluster number, in a way that would not be caused by a non-LFN aware file-system driver.

Internal Classes and Functions

You should never need to interact with these classes directly; use *FatFileSystem* (page 30) instead. These classes exist to enumerate and manipulate the FAT, and different types of root directory under FAT-12, FAT-16, and FAT-32, and sub-directories (which are common across FAT types).

class nobodd.fs.**FatTable**

Abstract `MutableSequence`[99] class representing the FAT table itself.

This is the basis for *Fat12Table* (page 34), *Fat16Table* (page 34), and *Fat32Table* (page 34). While all the implementations are potentially mutable (if the underlying memory mapping is writable), only direct replacement of FAT entries is valid. Insertion and deletion will raise `TypeError`[100].

[96] https://docs.python.org/3.12/library/io.html#io.RawIOBase
[97] https://docs.python.org/3.12/library/functions.html#open
[98] https://docs.python.org/3.12/library/functions.html#open

A concrete class is constructed by *FatFileSystem* (page 30) (based on the type of FAT format found). The *chain()* (page 34) method is used by *FatFile* (page 32) (and indirectly *FatSubDirectory* (page 37)) to discover the chain of clusters that make up a file (or sub-directory). The *free()* (page 34) method is used by writable *FatFile* (page 32) instances to find the next free cluster to write to. The *mark_free()* (page 34) and *mark_end()* (page 34) methods are used to mark a clusters as being free or as the terminal cluster of a file.

chain(*start*)

Generator method which yields all the clusters in the chain starting at *start*.

free()

Generator that scans the FAT for free clusters, yielding each as it is found. Iterating to the end of this generator raises OSError[101] with the code ENOSPC (out of space).

abstract get_all(*cluster*)

Returns the value of *cluster* in all copies of the FAT, as a tuple[102] (naturally, under normal circumstances, these should all be equal).

insert(*cluster*, *value*)

Raises TypeError[103]; the FAT length is immutable.

mark_end(*cluster*)

Marks *cluster* as the end of a chain. The value used to indicate the end of a chain is specific to the FAT size.

mark_free(*cluster*)

Marks *cluster* as free (this simply sets *cluster* to 0 in the FAT).

class nobodd.fs.**Fat12Table**(*mem*, *fat_size*, *info_mem=None*)

Concrete child of *FatTable* (page 33) for FAT-12 file-systems.

min_valid = 2

max_valid = 4079

end_mark = 4095

get_all(*cluster*)

Returns the value of *cluster* in all copies of the FAT, as a tuple[104] (naturally, under normal circumstances, these should all be equal).

class nobodd.fs.**Fat16Table**(*mem*, *fat_size*, *info_mem=None*)

Concrete child of *FatTable* (page 33) for FAT-16 file-systems.

min_valid = 2

max_valid = 65519

end_mark = 65535

get_all(*cluster*)

Returns the value of *cluster* in all copies of the FAT, as a tuple[105] (naturally, under normal circumstances, these should all be equal).

class nobodd.fs.**Fat32Table**(*mem*, *fat_size*, *info_mem=None*)

Concrete child of *FatTable* (page 33) for FAT-32 file-systems.

[99] https://docs.python.org/3.12/library/collections.abc.html#collections.abc.MutableSequence
[100] https://docs.python.org/3.12/library/exceptions.html#TypeError
[101] https://docs.python.org/3.12/library/exceptions.html#OSError
[102] https://docs.python.org/3.12/library/stdtypes.html#tuple
[103] https://docs.python.org/3.12/library/exceptions.html#TypeError
[104] https://docs.python.org/3.12/library/stdtypes.html#tuple
[105] https://docs.python.org/3.12/library/stdtypes.html#tuple

min_valid = 2

max_valid = 268435439

end_mark = 268435455

free()

Generator that scans the FAT for free clusters, yielding each as it is found. Iterating to the end of this generator raises `OSError`[106] with the code ENOSPC (out of space).

get_all(*cluster*)

Returns the value of *cluster* in all copies of the FAT, as a `tuple`[107] (naturally, under normal circumstances, these should all be equal).

class nobodd.fs.**FatClusters**(*mem*, *cluster_size*)

`MutableSequence`[108] representing the clusters of the file-system itself.

While the sequence is mutable, clusters cannot be deleted or inserted, only read and (if the underlying buffer is writable) re-written.

insert(*cluster*, *value*)

Raises `TypeError`[109]; the FS length is immutable.

property readonly

Returns `True`[110] if the underlying buffer is read-only.

property size

Returns the size (in bytes) of clusters in the file-system.

class nobodd.fs.**FatDirectory**

An abstract `MutableMapping`[111] representing a FAT directory[112]. The mapping is ostensibly from filename to *DirectoryEntry* (page 40) instances, but there are several oddities to be aware of.

In VFAT, many files effectively have *two* filenames: the original DOS "short" filename (SFN hereafter) and the VFAT "long" filename (LFN hereafter). All files have an SFN; any file may optionally have an LFN. The SFN is stored in the *DirectoryEntry* (page 40) which records details of the file (mode, size, cluster, etc). The optional LFN is stored in leading *LongFilenameEntry* (page 40) records.

Even when *LongFilenameEntry* (page 40) records do *not* precede a *DirectoryEntry* (page 40), the file may still have an LFN that differs from the SFN in case only, recorded by flags in the *DirectoryEntry* (page 40). Naturally, some files still only have one filename because the LFN doesn't vary in case from the SFN, e.g. the special directory entries "." and "..", and anything which conforms to original DOS naming rules like "README.TXT".

For the purposes of listing files, most FAT implementations (including this one) ignore the SFNs. Hence, iterating over this mapping will *not* yield the SFNs as keys (unless the SFN is equal to the LFN), and they are *not* counted in the length of the mapping. However, for the purposes of testing existence, opening, etc., FAT implementations allow the use of SFNs. Hence, testing for membership, or manipulating entries via the SFN will work with this mapping, and will implicitly manipulate the associated LFNs (e.g. deleting an entry via a SFN key will also delete the associated LFN key).

In other words, if a file has a distinct LFN and SFN, it has *two* entries in the mapping (a "visible" LFN entry, and an "invisible" SFN entry). Further, note that FAT is case retentive (for LFNs; SFNs are folded uppercase), but not case sensitive. Hence, membership tests and retrieval from this mapping are case insensitive with regard to keys.

Finally, note that the values in the mapping are always instances of *DirectoryEntry* (page 40). *LongFilenameEntry* (page 40) instances are neither accepted nor returned; these are managed internally.

[106] https://docs.python.org/3.12/library/exceptions.html#OSError
[107] https://docs.python.org/3.12/library/stdtypes.html#tuple
[108] https://docs.python.org/3.12/library/collections.abc.html#collections.abc.MutableSequence
[109] https://docs.python.org/3.12/library/exceptions.html#TypeError
[110] https://docs.python.org/3.12/library/constants.html#True

MAX_SFN_SUFFIX = 65535

_clean_entries()

Find and remove all deleted entries from the directory.

The method scans the directory for all directory entries and long filename entries which start with 0xE5, indicating a deleted entry, and overwrites them with later (not deleted) entries. Trailing entries are then zeroed out. The return value is the new offset of the terminal entry.

_get_names(*filename*)

Given a *filename*, generate an appropriately encoded long filename (encoded in little-endian UCS-2), short filename (encoded in the file-system's SFN encoding), extension, and the case attributes. The result is a 4-tuple: `lfn, sfn, ext, attr`.

`lfn`, `sfn`, and `ext` will be `bytes`[113] strings, and `attr` will be an `int`[114]. If *filename* is capable of being represented as a short filename only (potentially with non-zero case attributes), `lfn` in the result will be zero-length.

_get_unique_sfn(*prefix*, *ext*)

Given *prefix* and *ext*, which are `str`[115], of the short filename prefix and extension, find a suffix that is unique in the directory (amongst both long *and* short filenames, because these are still in the same namespace).

For example, in a directory containing `default.config` (which has shortname `DEFAUL~1.CON`), given the filename and extension `default.conf`, this function will return the `str`[116] `DEFAUL~2.CON`.

Because the search requires enumeration of the whole directory, which is expensive, an artificial limit of *MAX_SFN_SUFFIX* (page 35) is enforced. If this is reached, the search will terminate with an `OSError`[117] with code ENOSPC (out of space).

_group_entries()

Generator which yields an offset, and a sequence of either *LongFilenameEntry* (page 40) and *DirectoryEntry* (page 40) instances.

Each tuple yielded represents a single (extant, non-deleted) file or directory with its long-filename entries at the start, and the directory entry as the final element. The offset associated with the sequence is the offset of the *directory entry* (not its preceding long filename entries). In other words, for a file with three long-filename entries, the following might be yielded:

```
(160, [
    <LongFilenameEntry>),
    <LongFilenameEntry>),
    <LongFilenameEntry>),
    <DirectoryEntry>)
])
```

This indicates that the directory entry is at offset 160, preceded by long filename entries at offsets 128, 96, and 64.

abstract _iter_entries()

Abstract generator that is expected to yield successive offsets and the entries at those offsets as *DirectoryEntry* (page 40) instances or *LongFilenameEntry* (page 40) instances, as appropriate.

All instances must be yielded, in the order they appear on disk, regardless of whether they represent deleted, orphaned, corrupted, terminal, or post-terminal entries.

_join_lfn_entries(*entries*, *checksum*, *sequence=0*, *lfn=b''*)

Given *entries*, a sequence of *LongFilenameEntry* (page 40) instances, decode the long filename encoded within them, ensuring that all the invariants (sequence number, checksums, terminal flag, etc.) are obeyed.

Returns the decoded (str[118]) long filename, or None[119] if no valid long filename can be found. Emits various warnings if invalid entries are encountered during decoding, including *OrphanedLongFilename* (page 33) and *BadLongFilename* (page 33).

_prefix_entries(*filename*, *entry*)

Given *entry*, a *DirectoryEntry* (page 40), generate the necessary *LongFilenameEntry* (page 40) instances (if any), that are necessary to associate *entry* with the specified *filename*.

This function merely constructs the instances, ensuring the (many, convoluted!) rules are followed, including that the short filename, if one is generated, is unique in this directory, and the long filename is encoded and check-summed appropriately.

Note: The *filename* and *ext* fields of *entry* are ignored by this method. The only filename that is considered is the one explicitly passed in which becomes the basis for the long filename entries *and* the short filename stored within the *entry* itself.

The return value is the sequence of long filename entries and the modified directory entry in the order they should appear on disk.

_split_entries(*entries*)

Given *entries*, a sequence of *LongFilenameEntry* (page 40) instances, ending with a single *DirectoryEntry* (page 40) (as would typically be found in a FAT directory index), return the decoded long filename, short filename, and the directory entry record as a 3-tuple.

If no long filename entries are present, the long filename will be equal to the short filename (but may have lower-case parts).

Note: This function also carries out several checks, including the filename checksum, that all checksums match, that the number of entries is valid, etc. Any violations found may raise warnings including *OrphanedLongFilename* (page 33) and *BadLongFilename* (page 33).

---

abstract _update_entry(*offset*, *entry*)

Abstract method which is expected to (re-)write *entry* (a *DirectoryEntry* (page 40) or *LongFilenameEntry* (page 40) instance) at the specified *offset* in the directory.

items() → a set-like object providing a view on D's items

values() → an object providing a view on D's values

class nobodd.fs.**FatRoot**(*mem*, *encoding*)

An abstract derivative of *FatDirectory* (page 35) representing the (fixed-size) root directory of a FAT-12 or FAT-16 file-system. Must be constructed with *mem*, which is a buffer object covering the root directory clusters, and *encoding*, which is taken from *FatFileSystem.sfn_encoding* (page 31). The *Fat12Root* (page 37) and *Fat16Root* (page 38) classes are (trivial) concrete derivatives of this.

class nobodd.fs.**FatSubDirectory**(*fs*, *start*, *encoding*)

A concrete derivative of *FatDirectory* (page 35) representing a sub-directory in a FAT file-system (of any type). Must be constructed with *fs* (a *FatFileSystem* (page 30) instance), *start* (the first cluster of the sub-directory), and *encoding*, which is taken from *FatFileSystem.sfn_encoding* (page 31).

[111] https://docs.python.org/3.12/library/collections.abc.html#collections.abc.MutableMapping
[112] https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system#Directory_table
[113] https://docs.python.org/3.12/library/stdtypes.html#bytes
[114] https://docs.python.org/3.12/library/functions.html#int
[115] https://docs.python.org/3.12/library/stdtypes.html#str
[116] https://docs.python.org/3.12/library/stdtypes.html#str
[117] https://docs.python.org/3.12/library/exceptions.html#OSError
[118] https://docs.python.org/3.12/library/stdtypes.html#str
[119] https://docs.python.org/3.12/library/constants.html#None

class nobodd.fs.**Fat12Root**(*mem*, *encoding*)

Concrete, trivial derivative of *FatRoot* (page 37) which simply declares the root as belonging to a FAT-12 file-system.

fat_type = 'fat12'

class nobodd.fs.**Fat16Root**(*mem*, *encoding*)

Concrete, trivial derivative of *FatRoot* (page 37) which simply declares the root as belonging to a FAT-16 file-system.

fat_type = 'fat16'

class nobodd.fs.**Fat32Root**(*fs*, *start*, *encoding*)

This is a trivial derivative of *FatSubDirectory* (page 37) because, in FAT-32, the root directory is represented by the same structure as a regular sub-directory.

nobodd.fs.**fat_type**(*mem*)

Given a FAT[120] file-system at the start of the buffer *mem*, determine its type, and decode its headers. Returns a four-tuple containing:

- one of the strings "fat12", "fat16", or "fat32"
- a *BIOSParameterBlock* (page 38) instance
- a *ExtendedBIOSParameterBlock* (page 39) instance
- a *FAT32BIOSParameterBlock* (page 39), if one is present, or None[121] otherwise

nobodd.fs.**fat_type_from_count**(*bpb*, *ebpb*, *ebpb_fat32*)

Derives the type of the FAT[122] file-system when it cannot be determined directly from the *bpb* and *ebpb* headers (the *BIOSParameterBlock* (page 38), and *ExtendedBIOSParameterBlock* (page 39) respectively).

Uses known limits[123] on the number of clusters to derive the type of FAT in use. Returns one of the strings "fat12", "fat16", or "fat32".

6.2.2 nobodd.fat

Defines the data structures used by the FAT[124] file system. You should never need these directly; use the *nobodd.fs.FatFileSystem* (page 30) class instead.

Data Structures

class nobodd.fat.**BIOSParameterBlock**(*jump_instruction*, *oem_name*, *bytes_per_sector*, *sectors_per_cluster*, *reserved_sectors*, *fat_count*, *max_root_entries*, *fat16_total_sectors*, *media_descriptor*, *sectors_per_fat*, *sectors_per_track*, *heads_per_disk*, *hidden_sectors*, *fat32_total_sectors*)

A namedtuple()[125] representing the BIOS Parameter Block[126] found at the very start of a FAT file system (of any type). This provides several (effectively unused) legacy fields, but also several fields still used exclusively in later FAT variants (like the count of FAT-32 sectors).

classmethod from_buffer(*buf*, *offset=0*)

Construct a *BIOSParameterBlock* (page 38) from the specified *offset* (which defaults to 0) in the buffer protocol object, *buf*.

[120] https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system
[121] https://docs.python.org/3.12/library/constants.html#None
[122] https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system
[123] https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system#Size_limits
[124] https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system

classmethod from_bytes(*s*)

Construct a *BIOSParameterBlock* (page 38) from the byte-string *s*.

to_buffer(*buf*, *offset=0*)

Write this *BIOSParameterBlock* (page 38) to *buf*, a buffer protocol object, at the specified *offset* (which defaults to 0).

class nobodd.fat.**ExtendedBIOSParameterBlock**(*drive_number*, *extended_boot_sig*, *volume_id*, *volume_label*, *file_system*)

A namedtuple()[127] representing the Extended BIOS Parameter Block[128] found either immediately after the BIOS Parameter Block[129] (in FAT-12 and FAT-16 formats), or after the FAT32 BIOS Parameter Block[130] (in FAT-32 formats).

This provides several (effectively unused) legacy fields, but also provides the "file_system" field which is used as the primary means of distinguishing the different FAT types (see *nobodd.fs.fat_type()* (page 38)), and the self-explanatory "volume_label" field.

classmethod from_buffer(*buf*, *offset=0*)

Construct a *ExtendedBIOSParameterBlock* (page 39) from the specified *offset* (which defaults to 0) in the buffer protocol object, *buf*.

classmethod from_bytes(*s*)

Construct a *ExtendedBIOSParameterBlock* (page 39) from the byte-string *s*.

to_buffer(*buf*, *offset=0*)

Write this *ExtendedBIOSParameterBlock* (page 39) to *buf*, a buffer protocol object, at the specified *offset* (which defaults to 0).

class nobodd.fat.**FAT32BIOSParameterBlock**(*sectors_per_fat*, *mirror_flags*, *version*, *root_dir_cluster*, *info_sector*, *backup_sector*)

A namedtuple()[131] representing the FAT32 BIOS Parameter Block[132] found immediately after the BIOS Parameter Block[133] in FAT-32 formats. In FAT-12 and FAT-16 formats it should not occur.

This crucially provides the cluster containing the root directory (which is structured as a normal sub-directory in FAT-32) as well as the number of sectors per FAT, specifically for FAT-32. All other fields are ignored by this implementation.

classmethod from_buffer(*buf*, *offset=0*)

Construct a *FAT32BIOSParameterBlock* (page 39) from the specified *offset* (which defaults to 0) in the buffer protocol object, *buf*.

classmethod from_bytes(*s*)

Construct a *FAT32BIOSParameterBlock* (page 39) from the byte-string *s*.

to_buffer(*buf*, *offset=0*)

Write this *FAT32BIOSParameterBlock* (page 39) to *buf*, a buffer protocol object, at the specified *offset* (which defaults to 0).

class nobodd.fat.**FAT32InfoSector**(*sig1*, *reserved1*, *sig2*, *free_clusters*, *last_alloc*, *reserved2*, *sig3*)

A namedtuple()[134] representing the FAT32 Info Sector[135] typically found in the sector after the BIOS Parameter Block[136] in FAT-32 formats. In FAT-12 and FAT-16 formats it is not present.

This records the number of free clusters available, and the last allocated cluster, which can speed up the search for free clusters during allocation. Because this implementation is capable of writing, and thus allocating

[125] https://docs.python.org/3.12/library/collections.html#collections.namedtuple
[126] https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system#BIOS_Parameter_Block
[127] https://docs.python.org/3.12/library/collections.html#collections.namedtuple
[128] https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system#Extended_BIOS_Parameter_Block
[129] https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system#BIOS_Parameter_Block
[130] https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system#FAT32_Extended_BIOS_Parameter_Block
[131] https://docs.python.org/3.12/library/collections.html#collections.namedtuple
[132] https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system#FAT32_Extended_BIOS_Parameter_Block
[133] https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system#BIOS_Parameter_Block

clusters, and because the reserved fields must be ignored but not re-written, they are represented as strings here (rather than "x" NULs) to ensure they are preserved when writing.

classmethod from_buffer(*buf*, *offset=0*)

Construct a *FAT32InfoSector* (page 39) from the specified *offset* (which defaults to 0) in the buffer protocol object, *buf*.

classmethod from_bytes(*s*)

Construct a *FAT32InfoSector* (page 39) from the byte-string *s*.

to_buffer(*buf*, *offset=0*)

Write this *FAT32InfoSector* (page 39) to *buf*, a buffer protocol object, at the specified *offset* (which defaults to 0).

class nobodd.fat.**DirectoryEntry**(*filename*, *ext*, *attr*, *attr2*, *ctime_cs*, *ctime*, *cdate*, *adate*, *first_cluster_hi*, *mtime*, *mdate*, *first_cluster_lo*, *size*)

A namedtuple()[137] representing a FAT directory entry[138]. This is a fixed-size structure which repeats up to the size of a cluster within a FAT root or sub-directory.

It contains the (8.3 sized) filename of an entry, the size in bytes, the cluster at which the entry's data starts, the entry's attributes (which determine whether the entry represents a file or another sub-directory), and (depending on the format), the creation, modification, and access timestamps.

Entries may represent deleted items in which case the first character of the *filename* will be 0xE5. If the *attr* is 0x0F, the entry is actually a long-filename entry and should be converted to *LongFilenameEntry* (page 40). If *attr* is 0x10, the entry represents a sub-directory. See directory entry[139] for more details.

classmethod eof()

Make a directory entry from NUL bytes; this is used to signify the end of the directory in indexes.

classmethod from_buffer(*buf*, *offset=0*)

Construct a *DirectoryEntry* (page 40) from the specified *offset* (which defaults to 0) in the buffer protocol object, *buf*.

classmethod from_bytes(*s*)

Construct a *DirectoryEntry* (page 40) from the byte-string *s*.

classmethod iter_over(*buf*)

Iteratively yields successive *DirectoryEntry* (page 40) instances from the buffer protocol object, *buf*.

Note: This method is entirely dumb and does not check whether the yielded instances are valid; it is up to the caller to determine the validity of entries.

---

to_buffer(*buf*, *offset=0*)

Write this *DirectoryEntry* (page 40) to *buf*, a buffer protocol object, at the specified *offset* (which defaults to 0).

class nobodd.fat.**LongFilenameEntry**(*sequence*, *name_1*, *attr*, *checksum*, *name_2*, *first_cluster*, *name_3*)

A namedtuple()[140] representing a FAT long filename[141]. This is a variant of the FAT directory entry[142] where the *attr* field is 0x0F.

Several of these entries will appear before their corresponding *DirectoryEntry* (page 40), but will be in *reverse* order. A *checksum* is incorporated for additional verification, and a *sequence* number indicating the number of segments, and which one is "last" (first in the byte-stream, but last in character order).

[134] https://docs.python.org/3.12/library/collections.html#collections.namedtuple
[135] https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system#FS_Information_Sector
[136] https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system#BIOS_Parameter_Block
[137] https://docs.python.org/3.12/library/collections.html#collections.namedtuple
[138] https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system#Directory_entry
[139] https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system#Directory_entry

classmethod from_buffer(*buf*, *offset=0*)

Construct a *LongFilenameEntry* (page 40) from the specified *offset* (which defaults to 0) in the buffer protocol object, *buf*.

classmethod from_bytes(*s*)

Construct a *LongFilenameEntry* (page 40) from the byte-string *s*.

classmethod iter_over(*buf*)

Iteratively yields successive *LongFilenameEntry* (page 40) instances from the buffer protocol object, *buf*.

Note: This method is entirely dumb and does not check whether the yielded instances are valid; it is up to the caller to determine the validity of entries.

---

to_buffer(*buf*, *offset=0*)

Write this *LongFilenameEntry* (page 40) to *buf*, a buffer protocol object, at the specified *offset* (which defaults to 0).

Functions

These utility functions help decode certain fields within the aforementioned structure, or check that tentative contents are valid.

nobodd.fat.**lfn_checksum**(*sfn*, *ext*)

Calculate the expected long-filename checksum given the *filename* and *ext* byte-strings of the short filename (from the corresponding `Directoryentry`).

nobodd.fat.**lfn_valid**(*s*)

Returns `True`[143] if `str`[144] *s* only contains characters valid in a VFAT long filename. Almost every Unicode character is permitted with a few exceptions (angle brackets, wildcards, etc).

6.2.3 nobodd.path

Defines the *FatPath* (page 42) class, a Path-like class for interacting with directories and sub-directories in a *FatFileSystem* (page 30) instance. You should never need to construct this class directly; instead it should be derived from the *root* (page 31) attribute which is itself a *FatPath* (page 42) instance.

```
>>> from nobodd.disk import DiskImage
>>> from nobodd.fs import FatFileSystem
>>> img = DiskImage('test.img')
>>> fs = FatFileSystem(img.partitions[1].data)
>>> for p in fs.root.iterdir():
...     print(repr(p))
...
FatPath(<FatFileSystem label='TEST' fat_type='fat16'>, '/foo')
FatPath(<FatFileSystem label='TEST' fat_type='fat16'>, '/bar.txt')
FatPath(<FatFileSystem label='TEST' fat_type='fat16'>, '/setup.cfg')
FatPath(<FatFileSystem label='TEST' fat_type='fat16'>, '/baz')
FatPath(<FatFileSystem label='TEST' fat_type='fat16'>, '/adir')
FatPath(<FatFileSystem label='TEST' fat_type='fat16'>, '/BDIR')
```

[140] https://docs.python.org/3.12/library/collections.html#collections.namedtuple
[141] https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system#VFAT_long_file_names
[142] https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system#Directory_entry
[143] https://docs.python.org/3.12/library/constants.html#True
[144] https://docs.python.org/3.12/library/stdtypes.html#str

FatPath

class nobodd.path.**FatPath**(*fs*, **pathsegments*)

A `Path`[145]-like object representing a filepath within an associated *FatFileSystem* (page 30).

There is rarely a need to construct this class directly. Instead, instances should be obtained via the *root* (page 31) property of a *FatFileSystem* (page 30). If constructed directly, *fs* is a *FatFileSystem* (page 30) instance, and *pathsegments* is the sequence of strings to be joined with a path separator into the path.

Instances provide almost all the facilities of the `pathlib.Path`[146] class they are modeled after, including the crucial *open()* (page 44) method, *iterdir()* (page 43), *glob()* (page 42), and *rglob()* (page 45) for enumerating directories, *stat()* (page 45), *is_dir()* (page 42), and *is_file()* (page 43) for querying information about files, division for construction of new paths, and all the usual *name* (page 46), *parent* (page 46), *stem* (page 47), and *suffix* (page 47) attributes. When the *FatFileSystem* (page 30) is writable, then *unlink()* (page 45), *touch()* (page 45), *mkdir()* (page 44), *rmdir()* (page 45), and *rename()* (page 44) may also be used.

Instances are also comparable for the purposes of sorting, but only within the same *FatFileSystem* (page 30) instance (comparisons across file-system instances raise `TypeError`[147]).

exists()

Whether the path points to an existing file or directory:

```
>>> fs
<FatFileSystem label='TEST' fat_type='fat16'>
>>> (fs.root / 'foo').exists()
True
>>> (fs.root / 'fooo').exists()
False
```

glob(*pattern*)

Glob the given relative *pattern* in the directory represented by this path, yielding matching files (of any kind):

```
>>> fs
<FatFileSystem label='TEST' fat_type='fat16'>
>>> sorted((fs.root / 'nobodd').glob('*.py'))
[FatPath(<FatFileSystem label='TEST' fat_type='fat16'>, '/nobodd/__init__.
↪py'),
 FatPath(<FatFileSystem label='TEST' fat_type='fat16'>, '/nobodd/disk.py'),
 FatPath(<FatFileSystem label='TEST' fat_type='fat16'>, '/nobodd/fat.py'),
 FatPath(<FatFileSystem label='TEST' fat_type='fat16'>, '/nobodd/fs.py'),
 FatPath(<FatFileSystem label='TEST' fat_type='fat16'>, '/nobodd/gpt.py'),
 FatPath(<FatFileSystem label='TEST' fat_type='fat16'>, '/nobodd/main.py'),
 FatPath(<FatFileSystem label='TEST' fat_type='fat16'>, '/nobodd/mbr.py'),
 FatPath(<FatFileSystem label='TEST' fat_type='fat16'>, '/nobodd/tftp.py'),
 FatPath(<FatFileSystem label='TEST' fat_type='fat16'>, '/nobodd/tools.py
↪')]
```

Patterns are the same as for `fnmatch()`[148], with the addition of "**" which means "this directory and all subdirectories, recursively". In other words, it enables recurisve globbing.

> **Warning:** Using the "**" pattern in large directory trees may consume an inordinate amount of time.

is_absolute()

Return whether the path is absolute or not. A path is considered absolute if it has a "/" prefix.

is_dir()

Return a bool[149] indicating whether the path points to a directory. False[150] is also returned if the path doesn't exist.

is_file()

Returns a bool[151] indicating whether the path points to a regular file. False[152] is also returned if the path doesn't exist.

is_mount()

Returns a bool[153] indicating whether the path is a *mount point*. In this implementation, this is only True[154] for the root path.

is_relative_to(**other*)

Return whether or not this path is relative to the *other* path.

iterdir()

When the path points to a directory, yield path objects of the directory contents:

```
>>> fs
<FatFileSystem label='TEST' fat_type='fat16'>
>>> for child in fs.root.iterdir(): child
...
FatPath(<FatFileSystem label='TEST' fat_type='fat16'>, '/foo')
FatPath(<FatFileSystem label='TEST' fat_type='fat16'>, '/bar.txt')
FatPath(<FatFileSystem label='TEST' fat_type='fat16'>, '/setup.cfg')
FatPath(<FatFileSystem label='TEST' fat_type='fat16'>, '/baz')
FatPath(<FatFileSystem label='TEST' fat_type='fat16'>, '/adir')
FatPath(<FatFileSystem label='TEST' fat_type='fat16'>, '/BDIR')
```

The children are yielded in arbitrary order (the order they are found in the file-system), and the special entries `'.'` and `'..'` are not included.

joinpath(**other*)

Calling this method is equivalent to combining the path with each of the *other* arguments in turn:

```
>>> fs
<FatFileSystem label='TEST' fat_type='fat16'>
>>> fs.root
FatPath(<FatFileSystem label='TEST' fat_type='fat16'>, '/')
>>> fs.root.joinpath('nobodd')
FatPath(<FatFileSystem label='TEST' fat_type='fat16'>, '/nobodd')
>>> fs.root.joinpath('nobodd', 'main.py')
FatPath(<FatFileSystem label='TEST' fat_type='fat16'>, '/nobodd/main.py')
```

match(*pattern*)

Match this path against the provided glob-style pattern. Returns a bool[155] indicating if the match is successful.

If *pattern* is relative, the path may be either relative or absolute, and matching is done from the right:

```
>>> fs
<FatFileSystem label='TEST' fat_type='fat16'>
>>> f = fs / 'nobodd' / 'mbr.py'
>>> f
FatPath(<FatFileSystem label='TEST' fat_type='fat16'>, '/nobodd/mbr.py')
>>> f.match('*.py')
True
>>> f.match('nobodd/*.py')
True
>>> f.match('/*.py')
False
```

As FAT file-systems are case-insensitive, all matches are likewise case-insensitive.

mkdir(*mode=511*, *parents=False*, *exist_ok=False*)

Create a new directory at this given path. The *mode* parameter exists only for compatibility with `pathlib.Path`[156] and is otherwise ignored. If the path already exists, `FileExistsError`[157] is raised.

If *parents* is true, any missing parents of this path are created as needed.

If *parents* is false (the default), a missing parent raises `FileNotFoundError`[158].

If *exist_ok* is false (the default), `FileExistsError`[159] is raised if the target directory already exists.

If *exist_ok* is true, `FileExistsError`[160] exceptions will be ignored (same behavior as the POSIX `mkdir -p` command), but only if the last path component is not an existing non-directory file.

open(*mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*)

Open the file pointed to by the path, like the built-in `open()`[161] function does. The *mode*, *buffering*, *encoding*, *errors* and *newline* options are as for the `open()`[162] function. If successful, a *FatFile* (page 32) instance is returned.

Note: This implementation is read-only, so any modes other than "r" and "rb" will fail with `PermissionError`[163].

---

read_bytes()

Return the binary contents of the pointed-to file as a bytes object:

```
>>> fs
<FatFileSystem label='TEST' fat_type='fat16'>
>>> (fs.root / 'foo').read_text()
b'foo\n'
```

read_text(*encoding=None*, *errors=None*)

Return the decoded contents of the pointed-to file as a string:

```
>>> fs
<FatFileSystem label='TEST' fat_type='fat16'>
>>> (fs.root / 'foo').read_text()
'foo\n'
```

relative_to(**other*)

Compute a version of this path relative to the path represented by *other*. If it's impossible, `ValueError`[164] is raised.

rename(*target*)

Rename this file or directory to the given *target*, and return a new *FatPath* (page 42) instance pointing to target. If *target* exists and is a file, it will be replaced silently. *target* can be either a string or another path object:

```
>>> p = fs.root / 'foo'
>>> p.open('w').write('some text')
9
>>> target = fs.root / 'bar'
>>> p.rename(target)
FatPath(<FatFileSystem label='TEST' fat_type='fat16'>, '/bar')
>>> target.read_text()
'some text'
```

The target path must be absolute. There are no guarantees of atomic behaviour (in contrast to `os.rename()`[165]).

Note: `pathlib.Path.rename()`[166] permits relative paths, but interprets them relative to the working directory which is a concept *FatPath* (page 42) does not support.

resolve (*strict=False*)

Make the path absolute, resolving any symlinks. A new *FatPath* (page 42) object is returned.

`".."` components are also eliminated (this is the only method to do so). If the path doesn't exist and *strict* is `True`[167], `FileNotFoundError`[168] is raised. If *strict* is `False`[169], the path is resolved as far as possible and any remainder is appended without checking whether it exists.

Note that as there is no concept of the "current" directory within *FatFileSystem* (page 30), relative paths cannot be resolved by this function, only absolute.

rglob (*pattern*)

This is like calling *glob()* (page 42) with a prefix of "`**/`" to the specified *pattern*.

rmdir ()

Remove this directory. The directory must be empty.

stat (*, *follow_symlinks=True*)

Return a `os.stat_result`[170] object containing information about this path:

```
>>> fs
<FatFileSystem label='TEST' fat_type='fat16'>
>>> p = (fs.root / 'nobodd' / 'main.py')
>>> p.stat().st_size
388
>>> p.stat().st_ctime
1696606672.02
```

Note: In a FAT file-system, `atime` has day resolution, `mtime` has 2-second resolution, and `ctime` has either 2-second or millisecond resolution depending on the driver that created it. Directories have no timestamp information.

The *follow_symlinks* parameter is included purely for compatibility with `pathlib.Path.stat()`[171]; it is ignored as symlinks are not supported.

touch (*mode=438*, *exist_ok=True*)

Create a file at this given path. The *mode* parameter is only present for compatibility with `pathlib.Path`[172] and is otherwise ignored. If the file already exists, the function succeeds if *exist_ok* is `True`[173] (and its modification time is updated to the current time), otherwise `FileExistsError`[174] is raised.

unlink (*missing_ok=False*)

Remove this file. If the path points to a directory, use *rmdir()* (page 45) instead.

If *missing_ok* is `False`[175] (the default), `FileNotFoundError`[176] is raised if the path does not exist. If *missing_ok* is `True`[177], `FileNotFoundError`[178] exceptions will be ignored (same behaviour as the POSIX `rm -f` command).

with_name (*name*)

Return a new path with the *name* (page 46) changed. If the original path doesn't have a name, `ValueError`[179] is raised.

with_stem (*stem*)

Return a new path with the *stem* (page 47) changed. If the original path doesn't have a name, `ValueError`[180] is raised.

with_suffix (*suffix*)

Return a new path with the *suffix* (page 47) changed. If the original path doesn't have a suffix, the new *suffix* is appended instead. If the *suffix* is an empty string, the original suffix is removed.

write_bytes(*data*)

Open the file pointed to in bytes mode, write *data* to it, and close the file:

```
>>> p = fs.root / 'my_binary_file'
>>> p.write_bytes(b'Binary file contents')
20
>>> p.read_bytes()
b'Binary file contents'
```

An existing file of the same name is overwritten.

write_text(*data*, *encoding=None*, *errors=None*, *newline=None*)

Open the file pointed to in text mode, write *data* to it, and close the file:

```
>>> p = fs.root / 'my_text_file'
>>> p.write_text('Text file contents')
18
>>> p.read_text()
'Text file contents'
```

An existing file of the same name is overwritten. The optional parameters have the same meaning as in *open()* (page 44).

property anchor

Returns the concatenation of the drive and root. This is always "/".

property fs

Returns the *FatFileSystem* (page 30) instance that this instance was constructed with.

property name

A string representing the final path component, excluding the root:

```
>>> fs
<FatFileSystem label='TEST' fat_type='fat16'>
>>> p = (fs.root / 'nobodd' / 'main.py')
>>> p.name
'main.py'
```

property parent

The logical parent of the path:

```
>>> fs
<FatFileSystem label='TEST' fat_type='fat16'>
>>> p = (fs.root / 'nobodd' / 'main.py')
>>> p.parent
FatPath(<FatFileSystem label='TEST' fat_type='fat16'>, '/nobodd')
```

You cannot go past an anchor:

```
>>> p = (fs.root / 'nobodd' / 'main.py')
>>> p.parent.parent.parent
FatPath(<FatFileSystem label='TEST' fat_type='fat16'>, '/')
```

property parents

An immutable sequence providing access to the logical ancestors of the path:

```
>>> fs
<FatFileSystem label='TEST' fat_type='fat16'>
>>> p = (fs.root / 'nobodd' / 'main.py')
>>> p.parents
(FatPath(<FatFileSystem label='TEST' fat_type='fat16'>, '/nobodd'),
 FatPath(<FatFileSystem label='TEST' fat_type='fat16'>, '/'))
```

property parts

A tuple giving access to the path's various components:

```
>>> fs
<FatFileSystem label='TEST' fat_type='fat16'>
>>> p = (fs.root / 'nobodd' / 'main.py')
>>> p.parts
['/', 'nobodd', 'main.py']
```

property root

Returns a string representing the root. This is always "/".

property stem

The final path component, without its suffix:

```
>>> fs
<FatFileSystem label='TEST' fat_type='fat16'>
>>> p = (fs.root / 'nobodd' / 'main.py')
>>> p.stem
'main'
```

property suffix

The file extension of the final component, if any:

```
>>> fs
<FatFileSystem label='TEST' fat_type='fat16'>
>>> p = (fs.root / 'nobodd' / 'main.py')
>>> p.suffix
'.py'
```

property suffixes

A list of the path's file extensions:

```
>>> fs
<FatFileSystem label='TEST' fat_type='fat16'>
>>> p = (fs.root / 'nobodd.tar.gz')
>>> p.suffixes
['.tar', '.gz']
```

Internal Functions

nobodd.path.**get_cluster**(*entry*, *fat_type*)

Given *entry*, a *DirectoryEntry* (page 40), and the *fat_type* indicating the size of FAT clusters, return the first cluster of the file associated with the directory entry.

6.3 TFTP Service

The *nobodd.tftpd.TFTPBaseServer* (page 49) and *nobodd.tftpd.TFTPBaseHandler* (page 49) are two classes which may be customized to produce a TFTP server. Two example classes are included, *nobodd.tftpd.SimpleTFTPServer* (page 50) and *nobodd.tftpd.SimpleTFTPHandler* (page 49) which serve files directly from a specified path.

6.3.1 nobodd.tftpd

Defines several classes for the purposes of constructing TFTP servers. The most useful are *TFTPBaseHandler* (page 49) and *TFTPBaseServer* (page 49) which are abstract base classes for the construction of a TFTP server with an arbitrary source of files (these are used by nobodd's `main` module). In addition, `TFTPSimplerHandler` and `TFTPSimplerServer` are provided as a trivial example implementation of a straight-forward TFTP file server.

For example, to start a TFTP server which will serve files from the current directory on (unprivileged) port 1069:

```
>>> from nobodd.tftpd import SimpleTFTPServer
>>> server = SimpleTFTPServer(('0.0.0.0', 1069), '.')
>>> server.serve_forever()
```

[145] https://docs.python.org/3.12/library/pathlib.html#pathlib.Path
[146] https://docs.python.org/3.12/library/pathlib.html#pathlib.Path
[147] https://docs.python.org/3.12/library/exceptions.html#TypeError
[148] https://docs.python.org/3.12/library/fnmatch.html#fnmatch.fnmatch
[149] https://docs.python.org/3.12/library/functions.html#bool
[150] https://docs.python.org/3.12/library/constants.html#False
[151] https://docs.python.org/3.12/library/functions.html#bool
[152] https://docs.python.org/3.12/library/constants.html#False
[153] https://docs.python.org/3.12/library/functions.html#bool
[154] https://docs.python.org/3.12/library/constants.html#True
[155] https://docs.python.org/3.12/library/functions.html#bool
[156] https://docs.python.org/3.12/library/pathlib.html#pathlib.Path
[157] https://docs.python.org/3.12/library/exceptions.html#FileExistsError
[158] https://docs.python.org/3.12/library/exceptions.html#FileNotFoundError
[159] https://docs.python.org/3.12/library/exceptions.html#FileExistsError
[160] https://docs.python.org/3.12/library/exceptions.html#FileExistsError
[161] https://docs.python.org/3.12/library/io.html#io.open
[162] https://docs.python.org/3.12/library/io.html#io.open
[163] https://docs.python.org/3.12/library/exceptions.html#PermissionError
[164] https://docs.python.org/3.12/library/exceptions.html#ValueError
[165] https://docs.python.org/3.12/library/os.html#os.rename
[166] https://docs.python.org/3.12/library/pathlib.html#pathlib.Path.rename
[167] https://docs.python.org/3.12/library/constants.html#True
[168] https://docs.python.org/3.12/library/exceptions.html#FileNotFoundError
[169] https://docs.python.org/3.12/library/constants.html#False
[170] https://docs.python.org/3.12/library/os.html#os.stat_result
[171] https://docs.python.org/3.12/library/pathlib.html#pathlib.Path.stat
[172] https://docs.python.org/3.12/library/pathlib.html#pathlib.Path
[173] https://docs.python.org/3.12/library/constants.html#True
[174] https://docs.python.org/3.12/library/exceptions.html#FileExistsError
[175] https://docs.python.org/3.12/library/constants.html#False
[176] https://docs.python.org/3.12/library/exceptions.html#FileNotFoundError
[177] https://docs.python.org/3.12/library/constants.html#True
[178] https://docs.python.org/3.12/library/exceptions.html#FileNotFoundError
[179] https://docs.python.org/3.12/library/exceptions.html#ValueError
[180] https://docs.python.org/3.12/library/exceptions.html#ValueError

Handler Classes

class nobodd.tftpd.**TFTPBaseHandler**(*request*, *client_address*, *server*)

A abstract base handler for building TFTP servers.

Implements *do_RRQ()* (page 49) to handle the initial *RRQPacket* (page 54) of a transfer. This calls the abstract *resolve_path()* (page 49) to obtain the Path[181]-like object representing the requested file. Descendents must (at a minimum) override *resolve_path()* (page 49) to implement a TFTP server.

do_ERROR(*packet*)

Handles *ERRORPacket* (page 55) by ignoring it. The only way this should appear on the main port is at the start of a transfer, which would imply we're not going to start a transfer anyway.

do_RRQ(*packet*)

Handles *packet*, the initial *RRQPacket* (page 54) of a connection.

If option negotiation succeeds, and *resolve_path()* (page 49) returns a valid Path[182]-like object, this method will spin up a *TFTPSubServer* (page 52) instance in a background thread (see *TFTPSubServers* (page 52)) on an ephemeral port to handle all further interaction with this client.

resolve_path(*filename*)

Given *filename*, as requested by a TFTP client, returns a Path[183]-like object.

In the base class, this is an abstract method which raises NotImplementedError[184]. Descendents must override this method to return a Path[185]-like object, specifically one with a working open() [186] method, representing the file requested, or raise an OSError[187] (e.g. FileNotFoundError[188]) if the requested *filename* is invalid.

class nobodd.tftpd.**SimpleTFTPHandler**(*request*, *client_address*, *server*)

An implementation of *TFTPBaseHandler* (page 49) that overrides uses *SimpleTFTPServer.base_path* (page 50) for *resolve_path()* (page 49).

resolve_path(*filename*)

Resolves *filename* against *SimpleTFTPServer.base_path* (page 50).

Server Classes

class nobodd.tftpd.**TFTPBaseServer**(*address*, *handler_class*, *bind_and_activate=True*)

A abstract base for building TFTP servers.

To build a concrete TFTP server, make a descendent of *TFTPBaseHandler* (page 49) that overrides *resolve_path()* (page 49), then make a descendent of this class that calls super().__init__ with the overridden handler class. See *SimpleTFTPHandler* (page 49) and *SimpleTFTPServer* (page 50) for examples.

Note: While it is common to combine classes like UDPServer[189] and TCPServer[190] with the threading or fork-based mixins there is little point in doing so with *TFTPBaseServer* (page 49).

Only the initial packet of a TFTP transaction arrives on the "main" port; every packet after this is handled by a background thread with its own ephemeral port. Thus, multi-threading or multi-processing of the initial connection only applies to a single (minimal) packet.

---

[181] https://docs.python.org/3.12/library/pathlib.html#pathlib.Path
[182] https://docs.python.org/3.12/library/pathlib.html#pathlib.Path
[183] https://docs.python.org/3.12/library/pathlib.html#pathlib.Path
[184] https://docs.python.org/3.12/library/exceptions.html#NotImplementedError
[185] https://docs.python.org/3.12/library/pathlib.html#pathlib.Path
[186] https://docs.python.org/3.12/library/pathlib.html#pathlib.Path.open
[187] https://docs.python.org/3.12/library/exceptions.html#OSError
[188] https://docs.python.org/3.12/library/exceptions.html#FileNotFoundError

server_close()

Called to clean-up the server.

May be overridden.

class nobodd.tftpd.**SimpleTFTPServer**(*server_address*, *base_path*)

A trivial (pun intended) implementation of *TFTPBaseServer* (page 49) that resolves requested paths against *base_path* (a str[191] or Path[192]).

base_path

The *base_path* specified in the constructor.

Command Line Use

Just as http.server[193] can be invoked from the command line as a standalone server using the interpreter's -m[194] option, so *nobodd.tftpd* (page 48) can too. To serve the current directory as a TFTP server:

```
python -m nobodd.tftpd
```

The server listens to port 6969 by default. This is not the registered port 69 of TFTP, but as that port requires root privileges by default on UNIX platforms, a safer default was selected (the security provenance of this code is largely unknown, and certainly untested at higher privilege levels). The default port can be overridden by passed the desired port number as an argument:

```
python -m nobodd.tftpd 1069
```

By default, the server binds to all interfaces. The option -b/--bind specifies an address to which it should bind instead. Both IPv4 and IPv6 addresses are supported. For example, the following command causes the server to bind to localhost only:

```
python -m nobodd.tftpd --bind 127.0.0.1
```

By default, the server uses the current directory. The option -d/--directory specifies a directory from which it should serve files instead. For example:

```
python -m nobodd.tftpd --directory /tmp/
```

Internal Classes and Exceptions

The following classes and exceptions are entirely for internal use and should never be needed (directly) by applications.

class nobodd.tftpd.**TFTPClientState**(*address*, *path*, *mode='octet'*)

Represents the state of a single transfer with a client. Constructed with the client's *address* (format varies according to family), the *path* of the file to transfer (must be a Path[195]-like object, specifically one with a functioning open()[196] method), and the *mode* of the transfer (must be either *TFTP_BINARY* (page 53) or *TFTP_NETASCII* (page 54)).

address

The address of the client.

[189] https://docs.python.org/3.12/library/socketserver.html#socketserver.UDPServer
[190] https://docs.python.org/3.12/library/socketserver.html#socketserver.TCPServer
[191] https://docs.python.org/3.12/library/stdtypes.html#str
[192] https://docs.python.org/3.12/library/pathlib.html#pathlib.Path
[193] https://docs.python.org/3.12/library/http.server.html#module-http.server
[194] https://docs.python.org/3.12/using/cmdline.html#cmdoption-m

blocks

An internal mapping of block numbers to blocks. This caches blocks that have been read, transmitted, but not yet acknowledged. As `ACK` packets are received, blocks are removed from this cache.

block_size

The size, in bytes, of blocks to transfer to the client.

mode

The transfer mode. One of *TFTP_BINARY* (page 53) or *TFTP_NETASCII* (page 54).

source

The file-like object opened from the specified *path*.

timeout

The timeout, in nano-seconds, to use before re-transmitting packets to the client.

ack(*block_num*)

Specifies that *block_num* has been acknowledged by the client and can be removed from *blocks* (page 50), the internal block cache.

close()

Closes the source file associated with the client state. This method is idempotent.

get_block(*block_num*)

Returns the `bytes`[197] of the specified *block_num*.

If the *block_num* has not been read yet, this will cause the *source* (page 51) to be read. Otherwise, it will be returned from the as-yet unacknowledged block cache (in *blocks* (page 50)). If the block has already been acknowledged, which may happen asynchronously, this will raise *AlreadyAcknowledged* (page 53).

A `ValueError`[198] is raised if an invalid block is requested.

get_size()

Attempts to calculate the size of the transfer. This is used when negotiating the `tsize` option.

At first, `os.fstat()`[199] is attempted on the open file; if this fails (e.g. because there's no valid `fileno`), the routine will attempt to `seek()`[200] to the end of the file briefly to determine its size. Raises `OSError`[201] in the case that the size cannot be determined.

negotiate(*options*)

Called with *options*, a mapping of option names to values (both `str`[202]) that the client wishes to negotiate.

Currently supported options are defined in *nobodd.tftp.TFTP_OPTIONS* (page 54). The original *options* mapping is left unchanged. Returns a new options mapping containing only those options that we understand and accept, and with values adjusted to those that we can support.

Raises *BadOptions* (page 53) in the case that the client requests pathologically silly or dangerous options.

property finished

Indicates whether the transfer has completed or not. A transfer is considered complete when the final (under-sized) block has been sent to the client *and acknowledged*.

property transferred

Returns the number of bytes transferred to client and successfully acknowledged.

[195] https://docs.python.org/3.12/library/pathlib.html#pathlib.Path
[196] https://docs.python.org/3.12/library/pathlib.html#pathlib.Path.open
[197] https://docs.python.org/3.12/library/stdtypes.html#bytes
[198] https://docs.python.org/3.12/library/exceptions.html#ValueError
[199] https://docs.python.org/3.12/library/os.html#os.fstat
[200] https://docs.python.org/3.12/library/io.html#io.IOBase.seek
[201] https://docs.python.org/3.12/library/exceptions.html#OSError
[202] https://docs.python.org/3.12/library/stdtypes.html#str

class nobodd.tftpd.**TFTPHandler**(*request*, *client_address*, *server*)

Abstract base handler for TFTP transfers.

This handles decoding TFTP packets with the classes defined in *nobodd.tftp* (page 53). If the decoding is successful, it attempts to call a corresponding do_ method (e.g. *do_RRQ()* (page 49), *do_ACK()* (page 52)) with the decoded packet. The handler must return a *nobodd.tftp.Packet* (page 54) in response.

This base class defines no do_ methods itself; see *TFTPBaseHandler* (page 49) and *TFTPSubHandler* (page 52).

finish()

Overridden to send the response written to wfile. Returns the number of bytes written.

Note: In contrast to the usual DatagramRequestHandler, this method does *not* send an empty packet in the event that wfile has no content, as that confused several TFTP clients.

---

handle()

Attempts to decode the incoming *Packet* (page 54) and dispatch it to an appropriately named do_ method. If the method returns another *Packet* (page 54), it will be sent as the response.

setup()

Overridden to set up the rfile and wfile objects.

class nobodd.tftpd.**TFTPSubHandler**(*request*, *client_address*, *server*)

Handler for all client interaction after the initial *RRQPacket* (page 54).

Only the initial packet goes to the "main" TFTP port (69). After that, each transfer communicates between the client's original port (presumably in the ephemeral range) and an ephemeral server port, specific to that transfer. This handler is spawned by the main handler (a descendent of *TFTPBaseHandler* (page 49)) and deals with all further client communication. In practice this means it only handles *ACKPacket* (page 55) and *ERRORPacket* (page 55).

do_ACK(*packet*)

Handles *ACKPacket* (page 55) by calling *TFTPClientState.ack()* (page 51). Terminates the thread for this sub-handler if the transfer is complete, and otherwise sends the next *DATAPacket* (page 55) in response.

do_ERROR(*packet*)

Handles *ERRORPacket* (page 55) by terminating the transfer (in accordance with the spec.)

finish()

Overridden to note the last time we communicated with this client. This is used by the re-transmit algorithm.

handle()

Overridden to verify that the incoming packet came from the address (and port) that originally spawned this sub-handler. Logs and otherwise ignores all packets that do not meet this criteria.

class nobodd.tftpd.**TFTPSubServer**(*main_server*, *client_state*)

The server class associated with *TFTPSubHandler* (page 52).

You should never need to instantiate this class yourself. The base handler should create an instance of this to handle all communication with the client after the initial RRQ packet.

service_actions()

Overridden to handle re-transmission after a timeout.

class nobodd.tftpd.**TFTPSubServers**

Manager class for the threads running *TFTPSubServer* (page 52).

TFTPBaseServer (page 49) creates an instance of this to keep track of the background threads that are running transfers with *TFTPSubServer* (page 52).

add(*server*)

Add *server*, a *TFTPSubServer* (page 52) instance, as a new background thread to be tracked.

run()

Watches background threads for completed or otherwise terminated transfers. Shuts down all remaining servers (and their corresponding threads) at termination.

exception nobodd.tftpd.**TransferDone**

Exception raised internally to signal that a transfer has been completed.

exception nobodd.tftpd.**AlreadyAcknowledged**

Exception raised internally to indicate that a particular data packet was already acknowledged, and does not require repeated acknowlegement.

exception nobodd.tftpd.**BadOptions**

Exception raised when a client passes invalid options in a *RRQPacket* (page 54).

6.3.2 nobodd.tftp

Defines the data structures used by the Trivial File Transfer Protocol[203] (TFTP). You should never need these directly; use the classes in *nobodd.tftpd* (page 48) to construct a TFTP server instead.

Enumerations

class nobodd.tftp.**OpCode**(*value*)

Enumeration of op-codes for the Trivial File Transfer Protocol[204] (TFTP). These appear at the start of any TFTP packet to indicate what sort of packet it is.

class nobodd.tftp.**Error**(*value*)

Enumeration of error status for the Trivial File Transfer Protocol[205] (TFTP). These are used in packets with *OpCode* (page 53) ERROR to indicate the sort of error that has occurred.

Constants

nobodd.tftp.**TFTP_BLKSIZE**

nobodd.tftp.**TFTP_MIN_BLKSIZE**

nobodd.tftp.**TFTP_DEF_BLKSIZE**

nobodd.tftp.**TFTP_MAX_BLKSIZE**

Constants defining the blksize TFTP option; the name of the option, its minimum, default, and maximum values.

nobodd.tftp.**TFTP_TIMEOUT**

nobodd.tftp.**TFTP_UTIMEOUT**

nobodd.tftp.**TFTP_MIN_TIMEOUT_NS**

nobodd.tftp.**TFTP_DEF_TIMEOUT_NS**

nobodd.tftp.**TFTP_MAX_TIMEOUT_NS**

Constants defining the timeout and utimeout TFTP options; the name of the options, the minimum, default, and maximum values, in units of nano-seconds.

[203] https://en.wikipedia.org/wiki/Trivial_File_Transfer_Protocol
[204] https://en.wikipedia.org/wiki/Trivial_File_Transfer_Protocol
[205] https://en.wikipedia.org/wiki/Trivial_File_Transfer_Protocol

nobodd.tftp.**TFTP_BINARY**

nobodd.tftp.**TFTP_NETASCII**

nobodd.tftp.**TFTP_MODES**

Constants defining the available transfer modes.

nobodd.tftp.**TFTP_TSIZE**

Constant defining the name of the `tsize` TFTP option.

nobodd.tftp.**TFTP_OPTIONS**

Constant defining the TFTP options available for negotiation.

Packets

class nobodd.tftp.**Packet**

Abstract base class for all TFTP packets. This provides the class method *Packet.from_bytes()* (page 54) which constructs and returns the appropriate concrete sub-class for the *OpCode* (page 53) found at the beginning of the packet's data.

Instances of the concrete classes may be converted back to bytes[206] simply by calling bytes[207] on them:

```
>>> b = b'\x00\x01config.txt\0octet\0'
>>> r = Packet.from_bytes(b)
>>> r
RRQPacket(filename='config.txt', mode='octet', options=FrozenDict({}))
>>> bytes(r)
b'\x00\x01config.txt\x00octet\x00'
```

Concrete classes can also be constructed directly, for conversion into bytes[208] during transfer:

```
>>> bytes(ACKPacket(block=10))
b'\x00\x04\x00\n'
>>> bytes(RRQPacket('foo', 'netascii', {'tsize': 0}))
b'\x00\x01foo.txt\x00netascii\x00tsize\x000\x00'
```

classmethod from_bytes(*s*)

Given a bytes[209]-string *s*, checks the *OpCode* (page 53) at the front, and constructs one of the concrete packet types defined below, returning (instead of *Packet* (page 54) which is abstract):

```
>>> Packet.from_bytes(b'\x00\x01config.txt\0octet\0')
RRQPacket(filename='config.txt', mode='octet', options=FrozenDict({}))
```

classmethod from_data(*data*)

Constructs an instance of the packet class with the specified *data* (which is everything in the bytes[210]-string passed to *from_bytes()* (page 54) minus the header). This method is not implemented in *Packet* (page 54) but is expected to be implemented in any concrete descendant.

class nobodd.tftp.**RRQPacket**(*filename*, *mode*, *options=None*)

Concrete type for RRQ (read request) packets.

These packets are sent by a client to initiate a transfer. They include the *filename* to be sent, the *mode* to send it (one of the strings "octet" or "netascii"), and any *options* the client wishes to negotiate.

[206] https://docs.python.org/3.12/library/stdtypes.html#bytes
[207] https://docs.python.org/3.12/library/stdtypes.html#bytes
[208] https://docs.python.org/3.12/library/stdtypes.html#bytes
[209] https://docs.python.org/3.12/library/stdtypes.html#bytes
[210] https://docs.python.org/3.12/library/stdtypes.html#bytes

classmethod from_data(*data*)

Constructs an instance of the packet class with the specified *data* (which is everything in the bytes[211]-string passed to from_bytes() minus the header). This method is not implemented in *Packet* (page 54) but is expected to be implemented in any concrete descendant.

class nobodd.tftp.**WRQPacket**(*filename*, *mode*, *options=None*)

Concrete type for WRQ (write request) packets.

These packets are sent by a client to initiate a transfer to the server. They include the *filename* to be sent, the *mode* to send it (one of the strings "octet" or "netascii"), and any *options* the client wishes to negotiate.

class nobodd.tftp.**DATAPacket**(*block*, *data*)

Concrete type for DATA packets.

These are sent in response to RRQ, WRQ, or ACK packets and each contains a block of the file to transfer, *data* (by default, 512 bytes long unless this is the final DATA packet), and the *block* number.

classmethod from_data(*data*)

Constructs an instance of the packet class with the specified *data* (which is everything in the bytes[212]-string passed to from_bytes() minus the header). This method is not implemented in *Packet* (page 54) but is expected to be implemented in any concrete descendant.

class nobodd.tftp.**ACKPacket**(*block*)

Concrete type for ACK packets.

These are sent in response to DATA packets, and acknowledge the successful receipt of the specified *block*.

classmethod from_data(*data*)

Constructs an instance of the packet class with the specified *data* (which is everything in the bytes[213]-string passed to from_bytes() minus the header). This method is not implemented in *Packet* (page 54) but is expected to be implemented in any concrete descendant.

class nobodd.tftp.**ERRORPacket**(*error*, *message=None*)

Concrete type for ERROR packets.

These are sent by either end of a transfer to indicate a fatal error condition. Receipt of an ERROR packet immediately terminates a transfer without further acknowledgment.

The ERROR packet contains the *error* code (an *Error* (page 53) value) and a descriptive *message*.

classmethod from_data(*data*)

Constructs an instance of the packet class with the specified *data* (which is everything in the bytes[214]-string passed to from_bytes() minus the header). This method is not implemented in *Packet* (page 54) but is expected to be implemented in any concrete descendant.

class nobodd.tftp.**OACKPacket**(*options*)

Concrete type for OACK packets.

This is sent by the server instead of an initial DATA packet, when the client includes options in the RRQ packet. The content of the packet is all the *options* the server accepts, and their (potentially revised) values.

classmethod from_data(*data*)

Constructs an instance of the packet class with the specified *data* (which is everything in the bytes[215]-string passed to from_bytes() minus the header). This method is not implemented in *Packet* (page 54) but is expected to be implemented in any concrete descendant.

[211] https://docs.python.org/3.12/library/stdtypes.html#bytes
[212] https://docs.python.org/3.12/library/stdtypes.html#bytes
[213] https://docs.python.org/3.12/library/stdtypes.html#bytes
[214] https://docs.python.org/3.12/library/stdtypes.html#bytes
[215] https://docs.python.org/3.12/library/stdtypes.html#bytes

6.3.3 nobodd.netascii

Registers a Python codec to translate strings to the TFTP netascii encoding (defined in the TELNET RFC 764[216], under the printer and keyboard section). This is intended to translate line-endings of text files transparently between platforms, but only handles ASCII characters.

Note: TFTPd implementations could *probably* ignore this as a historical artefact at this point and assume all transfers will be done with "octet" (straight byte for byte) encoding, as seems to be common practice. However, netascii isn't terribly hard to support, hence the inclusion of this module.

The functions in this module should never need to be accessed directly. Simply use the 'netascii' encoding as you would any other Python byte-encoding:

```
>>> import os
>>> os.linesep
'\n'
>>> import nobodd.netascii
>>> 'foo\nbar\r'.encode('netascii')
b'foo\r\nbar\r\0'
>>> b'foo\r\nbar\r\0\r\r'.decode('netascii', errors='replace')
'foo\nbar\r??'
```

Internal Functions

nobodd.netascii.**encode**(*s*, *errors='strict'*, *final=False*)

Encodes the str[217] *s*, which must only contain valid ASCII characters, to the netascii bytes[218] representation.

The *errors* parameter specifies the handling of encoding errors in the typical manner ('strict', 'ignore', 'replace', etc). The *final* parameter indicates whether this is the end of the input. This only matters on the Windows platform where the line separator is 'rn' in which case a trailing 'r' character *may* be the start of a newline sequence.

The return value is a tuple of the encoded bytes[219] string, and the number of characters consumed from *s* (this may be less than the length of *s* when *final* is False[220]).

nobodd.netascii.**decode**(*s*, *errors='strict'*, *final=False*)

Decodes the bytes[221] string *s*, which must contain a netascii encoded string, to the str[222] representation (which can only contain ASCII characters).

The *errors* parameter specifies the handling of encoding errors in the typical manner ('strict', 'ignore', 'replace', etc). The *final* parameter indicates whether this is the end of the input. This matters as a trailing 'r' in the input is the beginning of a newline sequence, an encoded 'r', or an error (in other cases).

The return value is a tuple of the decoded str[223], and the number of characters consumed from *s* (this may be less than the length of *s* when *final* is False[224]).

class nobodd.netascii.**IncrementalEncoder**(*errors='strict'*)

Use codecs.iterencode()[225] to utilize this class for encoding:

[216] https://datatracker.ietf.org/doc/html/rfc764
[217] https://docs.python.org/3.12/library/stdtypes.html#str
[218] https://docs.python.org/3.12/library/stdtypes.html#bytes
[219] https://docs.python.org/3.12/library/stdtypes.html#bytes
[220] https://docs.python.org/3.12/library/constants.html#False
[221] https://docs.python.org/3.12/library/stdtypes.html#bytes
[222] https://docs.python.org/3.12/library/stdtypes.html#str
[223] https://docs.python.org/3.12/library/stdtypes.html#str
[224] https://docs.python.org/3.12/library/constants.html#False

```
>>> import os
>>> os.linesep
'\n'
>>> import nobodd.netascii
>>> import codecs
>>> it = ['foo', '\n', 'bar\r']
>>> b''.join(codecs.iterencode(it, 'netascii'))
b'foo\r\nbar\r\0'
```

class nobodd.netascii.**IncrementalDecoder**(*errors='strict'*)

Use `codecs.iterdecode()`[226] to utilize this class for encoding:

```
>>> import os
>>> os.linesep
'\n'
>>> import nobodd.netascii
>>> import codecs
>>> it = [b'foo\r', b'\n', b'bar\r', b'\0']
>>> ''.join(codecs.iterdecode(it, 'netascii'))
'foo\nbar\r'
```

class nobodd.netascii.**StreamWriter**(*stream*, *errors='strict'*)

encode(*s*, *errors='strict'*)

Encodes the object input and returns a tuple (output object, length consumed).

errors defines the error handling to apply. It defaults to 'strict' handling.

The method may not store state in the Codec instance. Use StreamWriter for codecs which have to keep state in order to make encoding efficient.

The encoder must be able to handle zero length input and return an empty object of the output object type in this situation.

reset()

Resets the codec buffers used for keeping internal state.

Calling this method should ensure that the data on the output is put into a clean state, that allows appending of new fresh data without having to rescan the whole stream to recover state.

class nobodd.netascii.**StreamReader**(*stream*, *errors='strict'*)

decode(*s*, *errors='strict'*, *final=False*)

Decodes the object input and returns a tuple (output object, length consumed).

input must be an object which provides the bf_getreadbuf buffer slot. Python strings, buffer objects and memory mapped files are examples of objects providing this slot.

errors defines the error handling to apply. It defaults to 'strict' handling.

The method may not store state in the Codec instance. Use StreamReader for codecs which have to keep state in order to make decoding efficient.

The decoder must be able to handle zero length input and return an empty object of the output object type in this situation.

[225] https://docs.python.org/3.12/library/codecs.html#codecs.iterencode
[226] https://docs.python.org/3.12/library/codecs.html#codecs.iterdecode

6.4 Command line applications

The *nobodd.server* (page 58) module contains the primary classes, *BootServer* (page 58) and *BootHandler* (page 58) which define a TFTP server (**nobodd-tftpd**) that reads files from FAT file-systems contained in OS images. The *nobodd.prep* (page 59) module contains the implementation of the **nobodd-prep** command, which customizes images prior to first net boot.

The *nobodd.config* (page 60) module provides configuration parsing facilities to these applications.

6.4.1 nobodd.server

This module contains the server and handler classes which make up the main **nobodd-tftpd** application, as well as the entry point for the application itself.

Handler Classes

class nobodd.server.**BootHandler**(*request*, *client_address*, *server*)

A descendent of *TFTPBaseHandler* (page 49) that resolves paths relative to the FAT file-system in the OS image associated with the Pi serial number which forms the initial directory.

resolve_path(*filename*)

Resolves *filename* relative to the OS image associated with the initial directory.

In other words, if the request is for `1234abcd/config.txt`, the handler will look up the board with serial number `1234abcd` in *BootServer.boards* (page 58), find the associated OS image, the FAT file-system within that image, and resolve `config.txt` within that file-system.

Server Classes

class nobodd.server.**BootServer**(*server_address*, *boards*)

A descendent of *TFTPBaseServer* (page 49) that is configured with *boards*, a mapping of Pi serial numbers to *Board* (page 61) instances, and uses *BootHandler* (page 58) as the handler class.

boards

The mapping of Pi serial numbers to *Board* (page 61) instances.

server_close()

Called to clean-up the server.

May be overridden.

Application Functions

nobodd.server.**main**(*args=None*)

The main entry point for the **nobodd-tftpd** application. Takes *args*, the sequence of command line arguments to parse. Returns the exit code of the application (0 for a normal exit, and non-zero otherwise).

If `DEBUG=1` is found in the application's environment, top-level exceptions will be printed with a full backtrace. `DEBUG=2` will launch PDB in port-mortem mode.

nobodd.server.**request_loop**(*server_address*, *boards*)

The application's request loop. Takes the *server_address* to bind to, which may be a `(address, port)` tuple, or an `int`[227] file-descriptor passed by a service manager, and the *boards* configuration, a `dict`[228] mapping serial numbers to *Board* (page 61) instances.

Raises *ReloadRequest* (page 59) or *TerminateRequest* (page 59) in response to certain signals, but is an infinite loop otherwise.

nobodd.server.**get_parser**()

Returns the command line parser for the application, pre-configured with defaults from the application's configuration file(s). See *ConfigArgumentParser()* (page 60) for more information.

Exceptions

exception nobodd.server.**ReloadRequest**

Exception class raised in *request_loop()* (page 58) to cause a reload. Handled in *main()* (page 58).

exception nobodd.server.**TerminateRequest**(*returncode*, *message=''*)

Exception class raised in *request_loop()* (page 58) to cause service termination. Handled in *main()* (page 58). Takes the return code of the application as the first argument.

6.4.2 nobodd.prep

This module contains the implementation (and entry point) of the **nobodd-prep** application.

Application Functions

nobodd.prep.**main**(*args=None*)

The main entry point for the **nobodd-prep** application. Takes *args*, the sequence of command line arguments to parse. Returns the exit code of the application (0 for a normal exit, and non-zero otherwise).

If `DEBUG=1` is found in the application's environment, top-level exceptions will be printed with a full backtrace. `DEBUG=2` will launch PDB in port-mortem mode.

nobodd.prep.**get_parser**()

Returns the command line parser for the application, pre-configured with defaults from the application's configuration file(s). See *ConfigArgumentParser()* (page 60) for more information.

nobodd.prep.**prepare_image**(*conf*)

Given the script's configuration in *conf*, an `argparse.Namespace`[229], resize the target image, and re-write the kernel command line within its boot partition to point to the configured NBD server and share.

nobodd.prep.**remove_items**(*fs*, *conf*)

In *fs*, a *FatFileSystem* (page 30), remove all items in the `list`[230] *conf.remove*, where *conf* is the script's configuration.

If any item is a directory, it and all files under it will be removed recursively. If an item in *to_remove* does not exist, a warning will be printed, but no error is raised.

nobodd.prep.**copy_items**(*fs*, *conf*)

Copy all `Path`[231] items in the `list`[232] *conf.copy* into *fs*, a *FatFileSystem* (page 30), where *conf* is the script's configuration.

If an item is a directory, it and all files under it will be copied recursively. If an item is a hard-link or a sym-link it will be copied as a regular file (since FAT does not support links). If an item does not exist, an `OSError`[233] will be raised. This is in contrast to `to_remove()` since it is assumed that control over the source file-system is under the caller's control, which is not the case in `to_remove()`.

[227] https://docs.python.org/3.12/library/functions.html#int
[228] https://docs.python.org/3.12/library/stdtypes.html#dict
[229] https://docs.python.org/3.12/library/argparse.html#argparse.Namespace
[230] https://docs.python.org/3.12/library/stdtypes.html#list
[231] https://docs.python.org/3.12/library/pathlib.html#pathlib.Path
[232] https://docs.python.org/3.12/library/stdtypes.html#list
[233] https://docs.python.org/3.12/library/exceptions.html#OSError

nobodd.prep.**rewrite_cmdline**(*fs*, *conf*)

Given the script's configuration *conf*, find the file *conf.cmdline* containing the kernel command-line in the *FatFileSystem* (page 30) *fs*, and re-write it to point the NBD share specified.

nobodd.prep.**detect_partitions**(*conf*)

Given the script's configuration in *conf*, an argparse.Namespace[234], open the target image, and attempt to detect the root and/or boot partition.

6.4.3 nobodd.config

This module contains the classes and functions used to configure the main nobodd application. These are not likely to be of much use to other applications, but are documented here just in case.

ConfigArgumentParser

class nobodd.config.**ConfigArgumentParser**(**args*, *template=None*, ***kwargs*)

A variant of ArgumentParser[235] that links arguments to specified keys in a ConfigParser[236] instance.

Typical usage is to construct an instance of *ConfigArgumentParser* (page 60), define the parameters and parameter groups on it, associating them with configuration section and key names as appropriate, then call *read_configs()* (page 61) to parse a set of configuration files. These will be checked against the (optional) *template* configuration passed to the initializer, which defines the set of valid sections and keys expected.

The resulting ConfigParser[237] forms the "base" configuration, prior to argument parsing. This can be optionally manipulated, before passing it to *set_defaults_from()* (page 61) to set the argument defaults. At this point, parse_args()[238] may be called to parse the command line arguments, knowing that defaults in the help will be drawn from the "base" configuration.

The resulting Namespace[239] object is the application's runtime configuration. For example:

```
>>> from pathlib import Path
>>> from nobodd.config import *
>>> parser = ConfigArgumentParser()
>>> tftp = parser.add_argument_group('tftp', section='tftp')
>>> tftp.add_argument('--listen', type=str, key='listen',
... help="the address on which to listen for connections "
... "(default: %(default)s)")
>>> Path('defaults.conf').write_text('''
... [tftp]
... listen = 127.0.0.1
... ''')
>>> defaults = parser.read_configs(['defaults.conf'])
>>> parser.set_defaults_from(defaults)
>>> parser.get_default('listen')
'127.0.0.1'
>>> config = parser.parse_args(['--listen', '0.0.0.0'])
>>> config.listen
'0.0.0.0'
```

Note that, after the call to *set_defaults_from()* (page 61), the parser's idea of the defaults has been drawn from the file-based configuration (and thus will be reflected in printed `--help`), but this is still overridden by the arguments passed to the command line.

add_argument(**args*, *section=None*, *key=None*, ***kwargs*)

Adds *section* and *key* parameters. These link the new argument to the specified configuration entry.

The default for the argument can be specified directly as usual, or can be read from the configuration (see *read_configs()* (page 61) and *set_defaults_from()* (page 61)). When arguments are parsed, the value assigned to this argument will be copied to the associated configuration entry.

[234] https://docs.python.org/3.12/library/argparse.html#argparse.Namespace

add_argument_group(*title=None*, *description=None*, *section=None*)

Adds a new argument group object and returns it.

The new argument group will likewise accept *section* and *key* parameters on its `add_argument()` (page 60) method. The *section* parameter will default to the value of the *section* parameter passed to this method (but may be explicitly overridden).

of_type(*type*)

Return a set of (section, key) tuples listing all configuration items which were defined as being of the specified *type* (with the *type* keyword passed to `add_argument()` (page 60).

read_configs(*paths*)

Constructs a `ConfigParser`[240] instance, and reads the configuration files specified by *paths*, a list of `Path`[241]-like objects, into it.

The method will check the configuration for valid section and key names, raising `ValueError`[242] on invalid items. It will also resolve any configuration values that have the type `Path`[243] relative to the path of the configuration file in which they were defined.

The return value is the configuration parser instance.

set_defaults_from(*config*)

Sets defaults for all arguments from their associated configuration entries in *config*.

update_config(*config*, *namespace*)

Copy values from *namespace* (an `argparse.Namespace`[244], presumably the result of calling something like `parse_args()`[245]) to *config*, a `ConfigParser`[246]. Note that namespace values will be converted to `str`[247] implicitly.

Board

class nobodd.config.**Board**(*serial*, *image*, *partition*, *ip*)

Represents a known board, recording its *serial* number, the *image* (filename) that the board should boot, the *partition* number within the *image* that contains the boot partition, and the IP address (if any) that the board should have.

classmethod from_section(*config*, *section*)

Construct a new `Board` (page 61) from the specified *section* of the *config* (a mapping, e.g. a `ConfigParser`[248] section).

classmethod from_string(*s*)

Construct a new `Board` (page 61) from the string *s* which is expected to be a comma-separated list of serial number, filename, partition number, and IP address. The last two parts (partition number and IP address) are optional and default to 1 and `None`[249] respectively.

[235] https://docs.python.org/3.12/library/argparse.html#argparse.ArgumentParser
[236] https://docs.python.org/3.12/library/configparser.html#configparser.ConfigParser
[237] https://docs.python.org/3.12/library/configparser.html#configparser.ConfigParser
[238] https://docs.python.org/3.12/library/argparse.html#argparse.ArgumentParser.parse_args
[239] https://docs.python.org/3.12/library/argparse.html#argparse.Namespace
[240] https://docs.python.org/3.12/library/configparser.html#configparser.ConfigParser
[241] https://docs.python.org/3.12/library/pathlib.html#pathlib.Path
[242] https://docs.python.org/3.12/library/exceptions.html#ValueError
[243] https://docs.python.org/3.12/library/pathlib.html#pathlib.Path
[244] https://docs.python.org/3.12/library/argparse.html#argparse.Namespace
[245] https://docs.python.org/3.12/library/argparse.html#argparse.ArgumentParser.parse_args
[246] https://docs.python.org/3.12/library/configparser.html#configparser.ConfigParser
[247] https://docs.python.org/3.12/library/stdtypes.html#str
[248] https://docs.python.org/3.12/library/configparser.html#configparser.ConfigParser
[249] https://docs.python.org/3.12/library/constants.html#None

Conversion Functions

nobodd.config.**port**(*s*)

Convert the string *s* into a port number. The string may either contain an integer representation (in which case the conversion is trivial, or a port name, in which case `socket.getservbyname()`[250] will be used to convert it to a port number (usually via NSS).

nobodd.config.**boolean**(*s*)

Convert the string *s* to a `bool`[251]. A typical set of case insensitive strings are accepted: "yes", "y", "true", "t", and "1" are converted to `True`[252], while "no", "n", "false", "f", and "0" convert to `False`[253]. Other values will result in `ValueError`[254].

nobodd.config.**size**(*s*)

Convert the string *s*, which must contain a number followed by an optional suffix (MB for mega-bytes, GB, for giga-bytes, etc.), and return the absolute integer value (scale the number in the string by the suffix given).

nobodd.config.**duration**(*s*)

Convert the string *s* to a `timedelta`[255]. The string must consist of white-space and/or comma separated values which are a number followed by a suffix indicating duration. For example:

```
>>> duration('1s')
timedelta(seconds=1)
>>> duration('5 minutes, 30 seconds')
timedelta(seconds=330)
```

The set of possible durations, and their recognized suffixes is as follows:

- *Microseconds*: microseconds, microsecond, microsec, micros, micro, useconds, usecond, usecs, usec, us, µseconds, µsecond, µsecs, µsec, µs
- *Milliseconds*: milliseconds, millisecond, millisec, millis, milli, mseconds, msecond, msecs, msec, ms
- *Seconds*: seconds, second, secs, sec, s
- *Minutes*: minutes, minute, mins, min, mi, m
- *Hours*: hours, hour, hrs, hr, h

If conversion fails, `ValueError`[256] is raised.

6.4.4 nobodd.systemd

This module contains a singleton class intended for communication with the *systemd(1)* service manager. It includes facilities for running a service as `Type=notify` where the service can actively communicate to systemd that it is ready to handle requests, is reloading its configuration, is shutting down, or that it needs more time to handle certain operations.

It also includes methods to ping the systemd watchdog, and to retrieve file-descriptors stored on behalf of the service (or provided as part of socket-activation).

[250] https://docs.python.org/3.12/library/socket.html#socket.getservbyname
[251] https://docs.python.org/3.12/library/functions.html#bool
[252] https://docs.python.org/3.12/library/constants.html#True
[253] https://docs.python.org/3.12/library/constants.html#False
[254] https://docs.python.org/3.12/library/exceptions.html#ValueError
[255] https://docs.python.org/3.12/library/datetime.html#datetime.timedelta
[256] https://docs.python.org/3.12/library/exceptions.html#ValueError

Systemd Class

class nobodd.systemd.**Systemd**(*address=None*)

Provides a simple interface to systemd's notification and watchdog services. It is suggested applications obtain a single, top-level instance of this class via *get_systemd()* (page 64) and use it to communicate with systemd.

available()

If systemd's notification socket is not available, raises RuntimeError[257]. Services expecting systemd notifications to be available can call this to assert that notifications will be noticed.

extend_timeout(*timeout*)

Notify systemd to extend the start / stop timeout by *timeout* seconds. A timeout will occur if the service does not call *ready()* (page 63) or terminate within *timeout* seconds but *only* if the original timeout (set in the systemd configuration) has already been exceeded.

For example, if the stopping timeout is configured as 90s, and the service calls *stopping()* (page 63), systemd expects the service to terminate within 90s. After 10s the service calls *extend_timeout()* (page 63) with a *timeout* of 10s. 20s later the service has not yet terminated but systemd does *not* consider the timeout expired as only 30s have elapsed of the original 90s timeout.

listen_fds()

Return file-descriptors passed to the service by systemd, e.g. as part of socket activation or file descriptor stores. It returns a dict[258] mapping each file-descriptor to its name, or the string "unknown" if no name was given.

main_pid(*pid=None*)

Report the main PID of the process to systemd (for services that confuse systemd with their forking behaviour). If *pid* is None, os.getpid()[259] is called to determine the calling process' PID.

notify(*state*)

Send a notification to systemd. *state* is a string type (if it is a unicode string it will be encoded with the 'ascii' codec).

ready()

Notify systemd that service startup is complete.

reloading()

Notify systemd that the service is reloading its configuration. Call *ready()* (page 63) when reload is complete.

stopping()

Notify systemd that the service is stopping.

watchdog_clean()

Unsets the watchdog environment variables so that no future child processes will inherit them.

> **Warning:** After calling this function, *watchdog_period()* (page 63) will return None but systemd will continue expecting *watchdog_ping()* (page 63) to be called periodically. In other words, you should call *watchdog_period()* (page 63) first and store its result somewhere before calling this function.

watchdog_period()

Returns the time (in seconds) before which systemd expects the process to call *watchdog_ping()* (page 63). If a watchdog timeout is not set, the function returns None.

watchdog_ping()

Ping the systemd watchdog. This must be done periodically if *watchdog_period()* (page 63) returns a value other than None.

watchdog_reset(*timeout*)

Reset the systemd watchdog timer to *timeout* seconds.

nobodd.systemd.**get_systemd**()

Return a single top-level instance of *Systemd* (page 63); repeated calls will return the same instance.

6.5 Miscellaneous

The *nobodd.tools* (page 64) module contains a variety of utility functions that either cross boundaries in the system or are entirely generic.

6.5.1 nobodd.tools

This module houses a series of miscellaneous functions which did not fit particularly well anywhere else and are needed across a variety of modules. They should never be needed by developers using nobodd as an application or a library, but are documented in case they are useful.

nobodd.tools.**labels**(*desc*)

Given the description of a C structure in *desc*, returns a tuple of the labels.

The str[260] *desc* must contain one entry per line (blank lines are ignored) where each entry consists of whitespace separated type (in Python struct[261] format) and label. For example:

```
>>> EBPB = '''
B     drive_number
1x    reserved
B     extended_boot_sig
4s    volume_id
11s   volume_label
8s    file_system
'''
>>> labels(EBPB)
('drive_number', 'extended_boot_sig', 'volume_id', 'volume_label',
'file_system')
```

Note the amount of whitespace is arbitrary, and further that any entries with the type "x" (which is used to indicate padding) will be excluded from the result ("reserved" is missing from the result tuple above).

The corresponding function *formats()* (page 64) can be used to obtain a tuple of the types.

nobodd.tools.**formats**(*desc*, *prefix='<'*)

Given the description of a C structure in *desc*, returns a concatenated str[262] of the types with an optional *prefix* (for endianness).

The str[263] *desc* must contain one entry per line (blank lines are ignored) where each entry consists of whitespace separated type (in Python struct[264] format) and label. For example:

```
>>> EBPB = '''
B     drive_number
1x    reserved
B     extended_boot_sig
4s    volume_id
11s   volume_label
8s    file_system
```

(continues on next page)

[257] https://docs.python.org/3.12/library/exceptions.html#RuntimeError
[258] https://docs.python.org/3.12/library/stdtypes.html#dict
[259] https://docs.python.org/3.12/library/os.html#os.getpid
[260] https://docs.python.org/3.12/library/stdtypes.html#str
[261] https://docs.python.org/3.12/library/struct.html#module-struct

(continued from previous page)

```
'''
>>> formats(EBPB)
'<B1xB4s11s8s'
```

Note the amount of whitespace is arbitrary, and further that any entries with the type "x" (which is used to indicate padding) are *not* excluded (unlike in *labels()* (page 64)).

The corresponding function *labels()* (page 64) can be used to obtain a tuple of the labels.

nobodd.tools.**get_best_family**(*host*, *port*)

Given a *host* name and a *port* specification (either a number or a service name), returns the network family (e.g. `socket.AF_INET`) and socket address to listen on as a tuple.

nobodd.tools.**format_address**(*address*)

Given a socket *address*, return a suitable `str`[265] representation of it.

Specifically, for IP4 addresses a simple "host:port" representation is used. For IP6 addresses (which typically incorporate ":" in the host portion), a "[host]:port" variant is used.

nobodd.tools.**pairwise**(*iterable*, */*)

Return an iterator of overlapping pairs taken from the input iterator.

s -> (s0,s1), (s1,s2), (s2, s3), …

nobodd.tools.**decode_timestamp**(*date*, *time*, *cs=0*)

Given the integers *date*, *time*, and optionally *cs* (from various fields in *DirectoryEntry* (page 40)), return a `datetime`[266] with the decoded timestamp.

nobodd.tools.**encode_timestamp**(*ts*)

Given a `datetime`[267], encode it as a FAT-compatible triple of three 16-bit integers representing (date, time, 1/100th seconds).

nobodd.tools.**any_match**(*s*, *expressions*)

Given a `str`[268] *s*, and *expressions*, a sequence of compiled regexes, return the `re.Match`[269] object from the first regex that matches *s*. If no regexes match, return `None`[270].

nobodd.tools.**exclude**(*ranges*, *value*)

Given a list non-overlapping of *ranges*, sorted in ascending order, this function modifies the range containing *value* (an integer, which must belong to one and only one range in the list) to exclude it.

class nobodd.tools.**BufferedTranscoder**(*stream*, *output_encoding*, *input_encoding=None*, *errors='strict'*)

A read-only transcoder, somewhat similar to `codecs.StreamRecoder`[271], but which strictly obeys the definition of the `read` method (with internal buffering).

This class is primarily intended for use in *netascii* (page 56) encoded transfers where it is used to transcode the underlying file stream into netascii encoding for the TFTP server.

The built-in `codecs.StreamRecoder`[272] class would seem ideal for this but for one issue: under certain circumstances (including those involved in netascii encoding), it violates the contract of the `read` method by returning *more* bytes than requested. For example:

[262] https://docs.python.org/3.12/library/stdtypes.html#str
[263] https://docs.python.org/3.12/library/stdtypes.html#str
[264] https://docs.python.org/3.12/library/struct.html#module-struct
[265] https://docs.python.org/3.12/library/stdtypes.html#str
[266] https://docs.python.org/3.12/library/datetime.html#datetime.datetime
[267] https://docs.python.org/3.12/library/datetime.html#datetime.datetime
[268] https://docs.python.org/3.12/library/stdtypes.html#str
[269] https://docs.python.org/3.12/library/re.html#re.Match
[270] https://docs.python.org/3.12/library/constants.html#None

```
>>> import io, codecs
>>> latin1_stream = io.BytesIO('abcdé'.encode('latin-1'))
>>> utf8_stream = codecs.StreamRecoder(latin1_stream,
... codecs.getencoder('utf-8'), codecs.getdecoder('utf-8'),
... codecs.getreader('latin-1'), codecs.getwriter('latin-1'))
>>> utf8_stream.read(3)
b'abc'
>>> utf8_stream.read(1)
b'd'
>>> utf8_stream.read(1)
b'\xc3\xa9'
```

This is alluded to in the documentation of `StreamReader.read` so it probably isn't a bug, but it is rather inconvenient when the caller is looking to fill a network packet of a specific size, and thus expects not to over-run.

This class implements a rather simpler recoder, which is read-only, does not permit seeking, but by use of an internal buffer, guarantees that the `read()` method (and associated methods like `readinto()`) will not return more bytes than requested.

It is constructed with the underlying *stream*, the name of the *output_encoding*, the name of the *input_encoding* (which defaults to the *output_encoding* when not specified), and the *errors* mode to use with the codecs. For example:

```
>>> import io
>>> from nobodd.tools import BufferedTranscoder
>>> latin1_stream = io.BytesIO('abcdé'.encode('latin-1'))
>>> utf8_stream = BufferedTranscoder(latin1_stream, 'utf-8', 'latin-1')
>>> utf8_stream.read(4)
b'abcd'
>>> utf8_stream.read(1)
b'\xc3'
>>> utf8_stream.read(1)
b'\xa9'
```

readable()

Return whether object was opened for reading.

If False, read() will raise OSError.

readall()

Read until EOF, using multiple read() call.

class nobodd.tools.**FrozenDict**(**args*)

A hashable, immutable mapping type.

The arguments to *FrozenDict* (page 66) are processed just like those to `dict`[273].

[271] https://docs.python.org/3.12/library/codecs.html#codecs.StreamRecoder
[272] https://docs.python.org/3.12/library/codecs.html#codecs.StreamRecoder
[273] https://docs.python.org/3.12/library/stdtypes.html#dict

CHAPTER

SEVEN

DEVELOPMENT

The main GitHub repository for the project can be found at:

https://github.com/waveform80/nobodd

7.1 Development installation

If you wish to develop nobodd, obtain the source by cloning the GitHub repository and then use the "develop" target of the Makefile which will install the package as a link to the cloned repository allowing in-place development. The following example demonstrates this method within a virtual Python environment:

```
$ sudo apt install build-essential git virtualenvwrapper
```

After installing `virtualenvwrapper` you'll need to restart your shell before commands like **`mkvirtualenv`** will operate correctly. Once you've restarted your shell, continue:

```
$ cd
$ mkvirtualenv nobodd
$ workon nobodd
(nobodd) $ git clone https://github.com/waveform80/nobodd.git
(nobodd) $ cd nobodd
(nobodd) $ make develop
```

To pull the latest changes from git into your clone and update your installation:

```
$ workon nobodd
(nobodd) $ cd ~/nobodd
(nobodd) $ git pull
(nobodd) $ make develop
```

To remove your installation, destroy the sandbox and the clone:

```
(nobodd) $ deactivate
$ rmvirtualenv nobodd
$ rm -rf ~/nobodd
```

7.2 Building the docs

If you wish to build the docs, you'll need a few more dependencies. Inkscape is used for conversion of SVGs to other formats, Graphviz is used for rendering certain charts, and TeX Live is required for building PDF output. The following command should install all required dependencies:

```
$ sudo apt install texlive-latex-recommended texlive-latex-extra \
    texlive-fonts-recommended texlive-xetex graphviz inkscape \
    python3-sphinx python3-sphinx-rtd-theme latexmk xindy
```

Once these are installed, you can use the "doc" target to build the documentation in all supported formats (HTML, ePub, and PDF):

```
$ workon nobodd
(nobodd) $ cd ~/nobodd
(nobodd) $ make doc
```

However, the easiest way to develop the documentation is with the "preview" target which will build the HTML version of the docs, and start a web-server to preview the output. The web-server will then watch for source changes (in both the documentation source, and the application's source) and rebuild the HTML automatically as required:

```
$ workon nobodd
(nobodd) $ cd ~/nobodd
(nobodd) $ make preview
```

The HTML output is written to `build/html` while the PDF output goes to `build/latex`.

7.3 Test suite

If you wish to run the nobodd test suite, follow the instructions in *Development installation* (page 67) above and then make the "test" target within the sandbox:

```
$ workon nobodd
(nobodd) $ cd ~/nobodd
(nobodd) $ make test
```

The test suite is also setup for usage with the **tox** utility, in which case it will attempt to execute the test suite with all supported versions of Python. If you are developing under Ubuntu you may wish to look into the Dead Snakes PPA[274] in order to install old/new versions of Python; the tox setup *should* work with the version of tox shipped with Ubuntu Focal, but more features (like parallel test execution) are available with later versions.

For example, to execute the test suite under tox, skipping interpreter versions which are not installed:

```
$ tox
```

To execute the test suite under all installed interpreter versions in parallel, using as many parallel tasks as there are CPUs, then displaying a combined report of coverage from all environments:

```
$ tox -p auto
$ coverage combine .coverage.py*
$ coverage report
```

[274] https://launchpad.net/~deadsnakes/%2Barchive/ubuntu/ppa

CHAPTER

EIGHT

CHANGELOG

8.1 Release 0.4 (2024-03-07)

- Use absolute paths for output of nbd-server and tftpd server configurations
- Include missing `#cloud-config` header in the tutorial

8.2 Release 0.3 (2024-03-06)

- Fix configuration reload when inheriting the TFTP socket from a service manager (#8[275])

8.3 Prototype 0.2 (unreleased)

- Add inheritance of the TFTP socket (#3[276])

8.4 Prototype 0.1 (unreleased)

- Initial tag

[275] https://github.com/waveform80/nobodd/issues/8
[276] https://github.com/waveform80/nobodd/issues/3

CHAPTER

NINE

LICENSE

PYTHON MODULE INDEX

n

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W