
Belay

Release 0.0.0

Brian Pugh

Mar 13, 2023

CONTENTS:

1	Who is Belay For?	3
2	What Problems Does Belay Solve?	5
3	Installation	7
4	Examples	9
4.1	Installation	9
4.2	Quick Start	10
4.3	CircuitPython	15
4.4	Connections	16
4.5	Package Manager	18
4.6	How Belay Works	24
4.7	API	26
5	Indices and tables	31
	Python Module Index	33
	Index	35

Belay is:

- A python library that enables the rapid development of projects that interact with hardware via a MicroPython or CircuitPython compatible board.
- A command-line tool for developing standalone MicroPython projects.
- A MicroPython package manager.

Belay supports wired serial connections (USB) and wireless connections via WebREPL over WiFi.

[Quick Video of Belay in 22 seconds.](#)

See [the documentation](#) for usage and other details.

WHO IS BELAY FOR?

Belay is for people creating a software project that needs to interact with hardware. Examples include:

- Control a motor so a webcam is always pointing at a person.
- Turn on an LED when you receive a notification.
- Read a potentiometer to control system volume.

The Belay Package Manager is for people that want to use public libraries, and get them on-device in an easy, repeatable, dependable manner.

WHAT PROBLEMS DOES BELAY SOLVE?

Typically, having a python script interact with hardware involves 3 major challenges:

1. On-device firmware (usually C or MicroPython) for directly handling hardware interactions. Typically this is developed, compiled, and uploaded as a (nearly) independent project.
2. A program on your computer that performs the tasks specified and interacts with the device.
3. Computer-to-device communication protocol. How are commands and results transferred? How does the device execute those commands?

This is lot of work if you just want your computer to do something simple like turn on an LED. Belay simplifies all of this by merging steps 1 and 2 into the same codebase, and manages step 3 for you. Code is automatically synced at the beginning of script execution.

The Belay Package Manager makes it easy to cache, update, and deploy third party libraries with your project.

INSTALLATION

Belay requires Python ≥ 3.8 and can be installed via:

```
pip install belay
```

The MicroPython-compatible board only needs MicroPython installed; no additional preparation is required. If using CircuitPython, and additional modification needs to be made to `boot.py`. See [documentation](#) for details.

EXAMPLES

Turning on an LED with Belay takes only 6 lines of code. Functions decorated with the `task` decorator are sent to the device and interpreted by the MicroPython interpreter. Calling the decorated function on-host sends a command to the device to execute the actual function.

```
import belay

device = belay.Device("/dev/ttyUSB0")

@device.task
def set_led(state):
    print(f"Printing from device; turning LED to {state}.")
    Pin(25, Pin.OUT).value(state)

set_led(True)
```

Outputs from `print` calls from on-device user-code are forwarded to host `stdout`.

For more examples, see the [examples](#) folder.

4.1 Installation

Belay requires Python ≥ 3.8 and can be installed from pypi via:

```
python -m pip install belay
```

To install directly from github, you can run:

```
python -m pip install git+https://github.com/BrianPugh/belay.git
```

For development, its recommended to use Poetry:

```
git clone https://github.com/BrianPugh/belay.git
cd belay
poetry install
```

4.2 Quick Start

Belay is a library that makes it quick and easy to interact with hardware via a MicroPython-compatible microcontroller. Belay has a single important class, `Device`:

```
import belay

device = belay.Device("/dev/ttyUSB0")
```

Creating a `Device` object connects to the board at the provided port. On connection, the device is reset into REPL mode, and a few common imports are performed on-device, namely:

```
import os, time, machine
from time import sleep
from micropython import const
from machine import ADC, I2C, Pin, PWM, SPI, Timer
```

The `Device` class has several useful methods:

1. `__call__` - Generic statement/expression string evaluation.
2. `setup` - Executes body on-device in a global context.
3. `task` - Executes function on-device.
4. `teardown` - Executes body on-device in a global context when connection is closed.
5. `thread` - Executes function on-device in a background thread.
6. `sync` - Generic file synchronization from host to device.
7. `sync_dependencies` - For python packages to sync bundled micropython dependencies to board.

These are described in more detail in the subsequent subsections.

4.2.1 call

Directly calling the device instance, like a function, invokes a python statement or expression on-device.

Invoking a python statement like:

```
ret = device("foo = 1 + 2")
```

would execute the code `foo = 1 + 2` on-device in the global context. Because this is a statement, the return value, `ret` is `None`.

Invoking a python expression like:

```
res = device("foo")
```

results in the return value `res == 3` on host.

4.2.2 setup

The `setup` decorator is a way of invoking code on-device in a global context, and is commonly used for imports and instantiating objects and hardware. For example:

```
@device.setup
def setup(pin_number):
    from machine import Pin

    led = Pin(pin_number)

setup(25)
```

is equivalent to:

```
device("pin_number = 25")
device("from machine import Pin")
device("led = Pin(pin_number)")
```

Functions decorated with `setup` should be called only a few times at most. For repeated functions calls, use the *task* decorator.

4.2.3 task

The `task` decorator sends the decorated function to the device, and replaces the host function with a remote-executor. Consider the following:

```
@device.task
def foo(a):
    return a * 2
```

Invoking `bar = foo(5)` on host sends a command to the device to execute the function `foo` with argument 5. The result, 10, is sent back to the host and results in `bar == 10`. This is the preferable way to interact with hardware.

Alternatively, the `foo` function will also be available at `device.task.foo`.

4.2.4 teardown

Same as `setup`, but automatically executes whenever `device.close()` is called. If `Device` is used as a context manager, `device.close()` is automatically called at context manager exit. Typically used for cleanup, like turning off LEDs or motors.

4.2.5 thread

`thread` is similar to `task`, but executes the decorated function in the background on-device.

```
@device.thread
def led_loop(period):
    led = Pin(25, Pin.OUT)
    while True:
        led.toggle()
        sleep(period)

led_loop(1.0) # Returns immediately
```

Not all MicroPython boards support threading, and those that do typically have a maximum of 1 thread. The decorated function has no return value.

4.2.6 sync

For more complicated hardware interactions, additional python modules/files need to be available on the device's filesystem. `sync` takes in a path to a local folder. The contents of the folder will be synced to the device's root directory.

For example, if the local filesystem looks like:

```
project
├── main.py
├── board
│   ├── foo.py
│   ├── bar
│   └── baz.py
```

Then, after `device.sync("board")` is ran from `main.py`, the remote filesystem will look like

```
foo.py
bar
└── baz.py
```

4.2.7 sync_dependencies

Syncs data that has been bundled with a python package. `sync_dependencies` is intended to make including micropython dependencies easier for pip-installable host-program.

```
from belay import Device

device = Device("/dev/ttyUSB0")

device.sync_dependencies("mypackage", "board")
# Alternative usage
import mypackage

device.sync_dependencies(mypackage, "board")
```


An intended use-case is to this method inconjunction with Belay's builtin package manager. Configure `dependencies_path` in `pyproject.toml` to point inside your python package, i.e. `dependencies_path="mypackage/dependencies"`. In doing so, micropython dependencies will be stored inside your package. For this example, lets assume that `pyproject.toml` defines main and dev dependencies. The data can then be synced:

```
device.sync_dependencies(mypackage, "dependencies/main", "dependencies/dev")
```

Depending on your build system, other non-belay configurations *may* need to be performed to ensure other data is included in your python package.

4.2.8 Subclassing Device

Device can be subclassed and have task/thread methods. Benefits of this approach is better organization, and being able to define tasks/threads before the actual object is instantiated.

Consider the following:

```
from belay import Device

device = Device("/dev/ttyUSB0")

@device.task
def foo(a):
    return a * 2
```

is roughly equivalent to:

```
from belay import Device

class MyDevice(Device):
    @Device.task
    def foo(a):
        return a * 2

device = MyDevice("/dev/ttyUSB0")
```

Marking methods as tasks/threads in a class requires using the capital `@Device.task` decorator. Methods marked with `@Device.task` are similar to `@staticmethod` in that they do **not** contain `self` in the method signature. To the device, each marked method is equivalent to an independent function. Methods can be marked with `@Device.setup` or `@Device.thread` for their respective functionality. Methods not marked with these decorators are just normal, boring python methods.

For methods decorated with `@Device.setup`, the flag `autoinit=True` can be set to automatically call the method at the end of object creation. The decorated method must have no parameters, otherwise a `ValueError` will be raised.

```
from belay import Device

class MyDevice(Device):
    @Device.setup(autoinit=True)
```

(continues on next page)

(continued from previous page)

```
def setup():
    foo = 42

device = MyDevice("/dev/ttyUSB0")
# Do NOT explicitly call `device.setup()`, it has already been invoked.
```

The Device class also has some hook methods that can be implemented to give customization to the object initialization process:

1. `__pre_autoinit__` - Called near the end of `__init__`, after convenience imports have been imported, but before methods marked with `@Device.setup(autoinit=True)` are invoked. This is a good location to sync additional micropython dependencies to device.
2. `__post_init__` - Called at the very end of `__init__`. This is a good location to set custom object attributes.

The following example will (in order):

1. Synchronize code located at `dependencies/main` within `my_package` to on-device `/lib`.
2. On-device, declare the global variable `operation_mode_pin` representing an input on pin 10.
3. On-host, query `operation_mode_pin` and set the attribute `operation_mode`, which could be used in other host methods.

```
from belay import Device

class MyDevice(Device):
    def __pre_autoinit__(self):
        # runs before `setup(autoinit=True)` decorated methods
        self.sync_dependencies("my_package", "dependencies/main")

    @Device.setup(autoinit=True)
    def setup():
        # A hypothetical jumper that controls how the device should function.
        operation_mode_pin = Pin(10, Pin.IN, Pin.PULL_UP)

    def __post_init__(self):
        # runs after `setup(autoinit=True)` decorated methods
        if self("operation_mode_pin.value"):
            self.operation_mode = "dev"
        else:
            self.operation_mode = "prod"
```

4.3 CircuitPython

Belay also supports [CircuitPython](#). Unlike MicroPython, CircuitPython automatically mounts the device's filesystem as a USB drive. This is usually convenient, but it makes the filesystem read-only to the python interpreter. To get around this, we need to manually add the following lines to `boot.py` on-device.

```
import storage

storage.remount("/")
```

Afterwards, reset the device and it's prepared for Belay.

4.3.1 Reverting

To revert this configuration, there are multiple options:

1. Edit `boot.py` using [Thonny](#), then reboot. Thonny (like Belay), operates via the REPL, so it has write-access since it's operating through circuitpython.
2. Through circuitpython's REPL via an interactive shell, such as `rshell` or `python -m serial.tools.miniterm`:

```
import os

os.remove("boot.py")
```

3. Using Belay in an interactive python prompt on host:

```
from belay import Device

device = Device("/dev/ttyUSB0") # replace with appropriate port
device("os.remove('boot.py')")
# Then reboot.
```

4.3.2 Physical Configuration

Storage mounting can be configured based on a physical pin state. Adding the following contents to `/boot.py` will configure the system to:

- Be in "normal" circuitpython mode if pin 14 is floating/high (due to configured pullup) on boot.
- Be in Belay-compatible mode if pin 14 is connected to ground on boot. This could be done, for example, by a push-button or a toggle switch.

```
import board
import storage
from digitalio import DigitalInOut, Pull

op_mode = DigitalInOut(board.GP14) # Choose whichever pin you would like
op_mode.switch_to_input(Pull.UP)

if not op_mode.value:
```

(continues on next page)

(continued from previous page)

```
# Mount system in host-readonly, circuitpython-writeable mode (Belay compatible).
storage.remount("/")
```

4.4 Connections

This section describes the connection with the device, as well as other elements regarding the connection of the device.

4.4.1 Reconnect

In the event that the device is temporarily disconnected, Belay can re-attempt to connect to the device and restore state. Typically, this will only work with projects that are purely sensor/actuator IO and do not have complicated internal states.

To enable this feature, set the keyword `attempts` when declaring your Device. Belay will attempt up to `attempts` times to reconnect to the device with 1 second delay in-between attempts. If Belay cannot restore the connection, it will raise a `ConnectionLost` exception.

Example:

```
device = Device("/dev/ttyUSB0", attempts=10)
```

By default, `attempts=0`, meaning that Belay will **not** attempt to reconnect with the device. If `attempts` is set to a negative value, Belay will infinitely attempt to reconnect with the device. If using a serial connection, a serial device `__might__` not be assigned to the name upon reconnecting. See the *UDev Rules* section for ways to ensure the same name is assigned upon reconnection.

How State is Restored

This section describes how the state is restored on-device, so the user can understand the limitations of Belay's reconnect feature.

1. When Belay sends a command to the device, the command is recorded into a command history. Function/generator calls **are not** recorded. These calls are expected to be frequent and not significantly modify the device's internal state.
2. On device disconnect, nothing happens.
3. On the next attempted Belay call, Belay will begin to attempt to reconnect with the device. This inherently resets the device, and consequently resets the device's python interpreter state.
4. Upon reconnection, **Belay will replay the entire call history**. For most projects, this should be relatively short and typically includes things like:
 - a. File-syncs: `device.sync("board/")`
 - b. Library imports: `device("import mysensor")`
 - b. Global object creations: `device("sensor = mysensor.Sensor()")`
 - c. Task definitions:

```
@device.task
def read_sensor():
    return sensor.read()
```

This history replay can result in a longer-than-expected blocking call.

4.4.2 Interface

Belay currently supports two connection interfaces:

1. Serial, typically over a USB cable. Recommended connection method.
2. WebREPL, typically over WiFi. Experimental and relatively slow due to higher command latency.

Serial

This is the typical connection method over a cable and is fairly self-explanatory.

UDev Rules

To ensure your serial device always connects with the same name, we can create a udev rule. Invoke `lsusb` to figure out some device information; the response should look like:

```
belay:~/belay$ lsusb
Bus 002 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 001 Device 003: ID 239a:80f4 Adafruit Pico
Bus 001 Device 002: ID 2109:3431 VIA Labs, Inc. Hub
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

Left of the colon is the 4-character `idVendor` value, and right of the colon is the 4-character `idProduct` value. Next, edit a file at `/etc/udev/rules.d/99-usb-serial.rules` to contain:

```
SUBSYSTEM=="tty", ATTRS{idVendor}=="xxxx", ATTRS{idProduct}=="yyyy", SYMLINK+="target"
```

For example, the following will map the "Adafruit Pico" to `/dev/ttyACM10`:

```
SUBSYSTEM=="tty", ATTRS{idVendor}=="239a", ATTRS{idProduct}=="80f4", SYMLINK+="ttyACM10"
```

Finally, the following command will reload the udev rules without having to do a reboot:

```
sudo udevadm control --reload-rules && sudo udevadm trigger
```

WebREPL

WebREPL is a protocol for accessing a MicroPython REPL over websockets.

WebREPL requires the MicroPython-bundled `webrepl` server running on-device. To run the WebREPL server on boot, we need two files on device:

1. `boot.py` that connects to your WiFi and starts the server.
2. `webrepl_cfg.py` that contains the password to access the WebREPL interface.

These files may look like (tested on an ESP32):

```
#####  
# boot.py #  
#####  
def do_connect(ssid, pwd):  
    import network  
  
    sta_if = network.WLAN(network.STA_IF)  
    if not sta_if.isconnected():  
        print("connecting to network...")  
        sta_if.active(True)  
        sta_if.connect(ssid, pwd)  
        while not sta_if.isconnected():  
            pass  
    print("network config:", sta_if.ifconfig())  
  
# Attempt to connect to WiFi network  
do_connect("YOUR WIFI SSID", "YOUR WIFI PASSWORD")  
  
import webrepl  
  
webrepl.start()
```

```
#####  
# webrepl_cfg.py #  
#####  
PASS = "python"
```

Once these files are on-device, connect to the device by providing the correct IP address and password. The `ws://` prefix tells Belay to use WebREPL.

```
device = belay.Device("ws://192.168.1.100", password="python")
```

4.5 Package Manager

The Belay CLI provides package manager functionality. At a high level, the Belay Package Manager does the following:

1. Reads settings from `pyproject.toml`. Dependencies are defined by URL's where they can be fetched. Commonly these are files hosted on github.
2. Downloads dependencies to the `.belay/dependencies/` folder. This folder should be committed to the project's git repository. This allows for repeatable deployment, even if a remote dependency goes missing or changes it's API.
3. Syncs the contents of `.belay/dependencies/` to the on-device `/lib` folder. This folder is included in the on-device `PATH`.
4. Syncs the contents of the project package directory.

4.5.1 Configuration

Belay's Package Manager uses `pyproject.toml` to define project configurations and dependencies. A typical project will look like:

```
[tool.belay]
name = "my_project_name"

[tool.belay.dependencies]
some_dependency = "https://github.com/BrianPugh/some-dependency/blob/main/some_
↳dependency.py"
```

Belay assumes the project contains a python-package (folder) with the same name as `tool.belay.name`. This directory is synced (in addition to dependencies) when `belay install` is ran.

Dependencies

To add python dependencies to a project, specify them in the `tool.belay.dependencies` section. This section contains a mapping of package names to URIs where they can be fetched from. There isn't a strong centralized micropython package repository, so Belay relies on directly specifying python file URLs. Belay supports several dependency values:

1. A string to a local file/folder path:

```
pathlib = "../micropython-lib/python-stdlib/pathlib/pathlib.py"
os = "../micropython-lib/python-stdlib/os/os"
```

2. A github link to a single file or a folder:

```
pathlib = "https://github.com/micropython/micropython-lib/blob/master/python-stdlib/
↳pathlib/pathlib.py"
os = "https://github.com/micropython/micropython-lib/tree/master/python-stdlib/os/os
↳"
```

3. A dictionary with a detailed specification:

```
pathlib = {uri="../micropython-lib/python-stdlib/pathlib/pathlib.py", develop=true}
```

4. A list of any of the above if multiple files are required for a single package:

```
os = [
    "https://github.com/micropython/micropython-lib/blob/master/python-stdlib/os/os/
↳__init__.py",
    "https://github.com/micropython/micropython-lib/blob/master/python-stdlib/os-
↳path/os/path.py",
]
```

This is most common for packages that have optional submodules.

Support for other types can be added. Please open up a github issue if Belay doesn't support a desired file source.

If specifying a dependency via dictionary, the following fields are available:

- `uri` - local or remote path to fetch data from. **Must** be provided.
- `develop` - Dependency is in "editable" mode. The dependency source is directly used during `belay install`. Primarily used for a local dependency actively under development. Defaults to `False`.

Groups

Belay supports groups of dependencies, allowing subsets of dependencies to be used in different situations. To declare a new dependency group, use a `tool.poetry.group.<group>` section where `<group>` is the name of a dependency group. `dev` is a common dependency group including packages like `unittest`.

```
[tool.belay.group.dev.dependencies]
unittest = [
    "https://github.com/micropython/micropython-lib/blob/master/python-stdlib/unittest/
    ↳unittest/__init__.py",
    "https://github.com/micropython/micropython-lib/blob/master/python-stdlib/unittest-
    ↳discover/unittest/__main__.py",
]
```

By default, all dependency groups are installed to device. A dependency group can be marked as optional, meaning it won't be installed during a `belay install` call unless explicitly specified `belay install --with=dev`.

```
[tool.belay.group.dev]
optional = true
```

All dependency groups are available to a host micropython interpreter via `belay run micropython`. See the *run* command section for more details.

Pytest

Since micropython and normal python code logic are mostly interoperable, code can be tested using `pytest` by adding the Belay dependency folder(s) to `pytest`'s configuration:

```
[tool.pytest.ini_options]
pythonpath = ".belay/dependencies/main .belay/dependencies/dev"
```

We recommend structuring projects to abstract hardware and micropython-specific features so that the majority of code can be tested with `pytest` using normal desktop CPython or `unittest` with desktop micropython. This will inherently produce better structured, more robust code and improve development iteration speed.

4.5.2 CLI Commands

This section describes all the commands available via `belay`. To get help from the command-line, add `--help` to any command for more information.

new

Creates a new directory structure suitable as a starting point for most `belay` projects.

```
belay new my-project
```

The project structure is as follows:

```
my-project/
├─ my-project/
│   └─ __init__.py
```

(continues on next page)

(continued from previous page)

```
└─ pyproject.toml
└─ README.md
```

update

Updates dependencies specified in `pyproject.toml`.

```
belay update
```

By default, the downloaded dependencies are stored in `.belay/dependencies/<group>/`. The `.belay/` folder **should be committed** to git and can be thought of as a dependency lock file.

Belay **will not** perform any dependency solving. It will only download the dependencies explicitly specified in the `pyproject.toml`. If a dependency itself has dependencies, they must be explicitly added to `pyproject.toml`.

By default, all dependencies are updated. To update only specific dependencies, list them as such:

```
belay update pathlib itertools
```

Previously downloaded dependencies that are no longer referenced in `tool.belay.dependencies` are automatically deleted.

install

Syncs the project and dependencies to device.

```
belay install [PORT]
```

To additionally sync a script to `/main.py`, specify the script using the `--main` option. After flashing, the device will be reset and the main script will execute.

```
belay install [PORT] --main main.py
```

The output of the main script can be monitored after flashing by including the `--follow` flag. Cancel the running script and exit the monitor via `ctrl-c`.

```
belay install [PORT] --main main.py --follow
```

During development, it is often convenient to specify a script to run without actually syncing it to `/main.py`. For this, specify the script using the `--run` option. The output will always be monitored.

```
belay install [PORT] --run main.py
```

To include a dependency group that has been declared optional, add the `--with` option.

```
belay install [PORT] --with dev
```

run

The `run` command serves 2 purposes:

1. Run a python script on-device.
2. Run a local executable in a pseudo-micropython-virtual-environment.

Running a Script on Device

When developing a script, it is often useful to edit it on-host and then execute it on-device. This helps circumvent issues with a flaky device filesystem. In the following command, `my_script.py` is executed on-device without explicitly writing it to the device's filesystem.

```
belay run [PORT] my_script.py
```

Virtual Environment

If the first argument after `run` is an executable, Belay will instead execute the remainder of the command after setting some environment variables. Namely, Belay will set the environment variable `MICROPYPATH` to all of the dependency groups' folders. This makes all of the dependencies accessible to a `micropython` binary, making it easier to test micropython code on-host.

```
belay run micropython my_script.py
```

This is not a true virtual environment; currently the `micropython` binary must be externally supplied.

clean

Removes any previously downloaded dependencies no longer specified in `tool.belay.dependencies`.

```
belay clean
```

`clean` is automatically invoked at the end of `belay update`, so this command will usually **not** be necessary.

cache

Belay keeps a cache of files that aid when downloading and updating dependencies. The location of this cache depends on the operating system:

- Windows: `%LOCALAPPDATA%\belay`
- MacOS: `~/Library/Caches/belay`
- Linux: `~/.cache/belay`

info

Displays Belay's cache location and other metadata.

```
$ belay cache info
Location: /Users/brianpugh/Library/Caches/belay
Elements: 1
Total Size: 3.84MB
```

list

Lists all the items Belay is currently caching.

```
$ belay cache list
git-github-micropython-micropython-lib
```

clear

Deletes all cached items that begin with the provided prefix

```
belay cache clear
```

For example, to delete all `git` caches, use the command:

```
belay cache clear git
```

To clear **all** caches, specify the `--all` flag.

```
belay cache clear --all
```

By default, Belay will display an interactive prompt to confirm the clearing action. This confirmation prompt can be bypassed by specifying the `--yes` flag.

```
belay cache clear --all --yes
```

terminal

Opens up an interactive terminal with the device. Press `ctrl-]` to exit the terminal.

```
belay terminal [PORT]
```

4.5.3 Q&A

How does Belay's package manager compare to mip?

Mip and Belay have different design goals and associated restrictions. Mip is designed to be ran on micropython, and is thusly restricted by available libraries. Belay is designed to be ran on full desktop python (e.g. cpython) to provide support to a micropython environment. The closest tool to Belay's Package Manager would actually be `mpremote mip`. With this tool you can specify remote files via a json configuration file.

Belay aims to provide a more robust, friendly experience by the following:

1. Use the standard `pyproject.toml` file for configurations and dependency specifications.
2. Make project robust to third-party changes by caching dependencies in-project. Your project won't become non-functional due to a remote dependency gone missing. Your project won't unexpectedly break due to a dependency change unless `belay update` is ran to update dependencies. Changes can be easily reverted due to git versioning.
3. Options to minify or compile code prior to sending it to device. This encourages more comments and docstrings.

What limitations does Belay's package manager have?

- Belay currently does not currently support `manifest.py`, but that may change in the future.
- Dependencies are not recursively searched/solved; if a dependency has it's own dependencies, you must add them yourself to your `pyproject.toml`.

Why should I commit .belay to my git repository?

The `.belay/` folder primarily contains cached micropython dependencies.

Cached dependencies are to be included in your git repo because:

1. Micropython libraries are inherently small due to their operating conditions. Adding them to the git repo is not an unreasonable burden.
2. The project will continue to work, even if an upstream dependency goes missing.
3. A lot of micropython libraries don't implement versioning, so more complicated dependency solving isn't feasible. Caching "known working" versions is the only convenient way of guaranteeing a repeatable deployment.

4.6 How Belay Works

In a nutshell, Belay sends python code (plain text) over the serial connection to the device's MicroPython Interactive Interpreter Mode (REPL) and parses back the response.

The easiest way to explain it is to walk through what's going under the hood with an example.

4.6.1 Device Creation

```
device = belay.Device("/dev/ttyUSB0")
```

This creates a Device object that connects to the microcontroller. Belay resets it, enters REPL mode, and then runs some convenience imports on the board.

4.6.2 Task - Sending Code Over

Consider the following decorated function:

```
@device.task
def set_led(state):
    """This function sets a pin to the specified state."""
    Pin(25, Pin.OUT).value(state) # Set a pin as an output, and set its value
```

The task decorator inspects the actual code of the function its decorating and sends it over to the microcontroller. Prior to sending the code over, a few preprocessing steps are required. At first, the code looks like:

```
def set_led(state):
    """This function sets a pin to the specified state."""
    Pin(25, Pin.OUT).value(state) # Set a pin as an output, and set its value
```

Belay can only send around 25,600 characters a second, so we want to reduce the amount of unnecessary characters. Some minification is performed to reduce the number of characters we have to send over to the device. The minification removes docstrings, comments, and unnecessary whitespace. Dont hesitate to add docstrings and comments to your code, they'll be stripped away before they reach your microcontroller. The minification maintains all variable names and line numbers, which can be important for debugging. After minification, the code looks like:

```
def set_led(state):
    0
    Pin(25,Pin.OUT).value(state)
```

The 0 is just a one character way of saying pass, in case the removed docstring was the entire body. This reduces the number of transmitted characters from 158 to just 53, offering a 3x speed boost.

After minification, the @__belay decorator is added. On-device, this defines a variant of the function, _belay_FUNCTION_NAME that performs the following actions:

1. Takes the returned value of the function, and serializes it to a string using repr.
2. Prints the resulting string to stdout, so it can be read by the host computer and deserialized via ast.literal_eval.

Conceptually, its as if the following code ran on-device (minification removed for clarity):

```
def set_led(state):
    Pin(25, Pin.OUT).value(state)

def _belay_set_led(*args, **kwargs):
    res = set_led(*args, **kwargs)
    print("_BELAYR" + repr(res))
```

A separate private function is defined with this serialization in case another on-device function calls set_led.

4.6.3 Task - Executing Function

Now that the function has been sent over and parsed by the microcontroller, we would like to execute it. The `@task` decorator returns a function that when invoked, creates and sends a command to the device, and then parses back the response. The complete lifecycle looks like this:

1. `set_led(True)` is called on the host. This doesn't execute the function we defined on host. Instead it triggers the following actions.
2. Belay creates the string `"_belay_set_led(True)"`.
3. Belay sends this command over serial to the REPL, causing it to execute on-device.
4. On-device, the result of `set_led(True)` is `None`. This gets serialized to the string `None`, which gets printed to stdout.
5. Belay reads this response from stdout, and deserializes it back to the `None` object.
6. `None` is returned on host from the `set_led(True)` call.

This has a few limitations, namely:

1. Each passed in argument must be a python literals (`None`, booleans, bytes, numbers, strings, sets, lists, and dicts).
2. User code cannot print a message that begins with `_BELAY`, otherwise the remainder of the message will attempt to be parsed.
3. The returned data of the function must also be a python literal(s).

4.7 API

4.7.1 Device

class `Device(*args, **kwargs)`

Belay interface into a micropython device.

Can be used as a context manager; calls `self.close` on exit.

Inherits from `autoregistry.Registry` for easy access to subclasses.

implementation

Implementation details of device.

Type

Implementation

MAX_CMD_HISTORY_LEN = 1000

close() → `None`

Close the connection to device.

Automatically called on context manager exit.

reconnect(*attempts: int | None = None*) → `None`

Reconnect to the device and replay the command history.

Parameters

attempts (*int*) -- Number of times to attempt to connect to board with a 1 second delay in-between. If `None`, defaults to whatever value was supplied to `init`. If `init` value is 0, then defaults to 1.

static setup() → Callable[[P], R]

static setup(kwargs)** → Callable[[Callable[[P], R]], Callable[[P], R]]

Execute decorated function's body in a global-context on-device when called.

Function arguments are also set in the global context.

Can either be used as a staticmethod `@Device.setup` for marking methods in a subclass of `Device`, or as a standard method `@device.setup` for marking functions to a specific `Device` instance.

Parameters

- **f** (*Callable*) -- Function to decorate. Can only accept and return python literals.
- **minify** (*bool*) -- Minify cmd code prior to sending. Defaults to `True`.
- **register** (*bool*) -- Assign an attribute to `self.setup` with same name as `f`. Defaults to `True`.
- **record** (*bool*) -- Each invocation of the executer is recorded for playback upon reconnect. Defaults to `True`.
- **autoinit** (*bool*) -- Automatically invokes decorated functions at the end of object `__init__`. Methods will be executed in order-registered. Defaults to `False`.

soft_reset()

Reset device, executing `main.py` if available.

sync(*folder: str | Path, dst: str = '/', keep: None | list | str | bool = None, ignore: None | list | str = None, minify: bool = True, mpy_cross_binary: str | Path | None = None, progress_update=None*) → `None`

Sync a local directory to the remote filesystem.

For each local file, check the remote file's hash, and transfer if they differ. If a file/folder exists on the remote filesystem that doesn't exist in the local folder, then delete it (unless it's in `keep`).

Parameters

- **folder** (*str, Path*) -- Single file or directory of files to sync to the root of the board's filesystem.
- **dst** (*str*) -- Destination **directory** on device. Defaults to unpacking folder to root.
- **keep** (*None | str | list | bool*) -- Do NOT delete these file(s) on-device if not present in folder. If `true`, don't delete any files on device. If `false`, delete all unsynced files (same as passing []). If `dst` is `None`, defaults to `["boot.py", "webrepl_cfg.py", "lib"]`.
- **ignore** (*None | str | list*) -- Git's wildmatch patterns to NOT sync to the device. Defaults to `["*.pyc", "__pycache__", ".DS_Store", ".pytest_cache"]`.
- **minify** (*bool*) -- Minify python files prior to syncing. Defaults to `True`.
- **mpy_cross_binary** (*Union[str, Path, None]*) -- Path to mpy-cross binary. If provided, `.py` will automatically be compiled. Takes precedence over minifying.
- **progress_update** -- Partial for `rich.progress.Progress.update(task_id, ...)` to update with sync status.

sync_dependencies(*package: module | str, *subfolders: str | Path, dst='/lib', **kwargs*)

Convenience method for syncing dependencies bundled with package.

If using Belay's package manager feature, set `dependencies_path` to a folder *inside* your python package (e.g. `dependencies_path="mypackage/dependencies"`).

The following example will sync all the files/folders in `mypackage/dependencies/main` to device's `/lib`.

```
import mypackage

device.sync_package(mypackage, "dependencies/main")
```

For intended use, `sync_dependencies` should be **only be called once**. Multiple invocations overwrite/delete previous calls' contents.

```
# Good
device.sync_package(mypackage, "dependencies/main", "dependencies/dev")

# Bad (deletes on-device files from "dependencies/main")
device.sync_package(mypackage, "dependencies/main")
device.sync_package(mypackage, "dependencies/dev")
```

Parameters

- **package** (*Union[ModuleType, str]*) -- Either the imported package or the name of a package that contains the data we would like to sync.
- ***subfolders** -- Subfolder(s) to combine and then sync to dst. Typically something like "dependencies/main"
- **dst** (*Union[str, Path]*) -- On-device destination directory. Defaults to `/lib`.
- ****kwargs** -- Passed along to `Device.sync`.

static task() → *Callable[[P], R]*

static task() → *Callable[[Callable[[P], R]], Callable[[P], R]]*

Execute decorated function on-device.

Sends source code to device at decoration time. Execution sends involves much smaller overhead.

Can either be used as a staticmethod `@Device.task` for marking methods in a subclass of `Device`, or as a standard method `@device.task` for marking functions to a specific `Device` instance.

Parameters

- **f** (*Callable*) -- Function to decorate. Can only accept and return python literals.
- **minify** (*bool*) -- Minify cmd code prior to sending. Defaults to `True`.
- **register** (*bool*) -- Assign an attribute to `self.task` with same name as `f`. Defaults to `True`.
- **record** (*bool*) -- Each invocation of the executer is recorded for playback upon reconnect. Only recommended to be set to `True` for a setup-like function. Defaults to `False`.

static teardown() → *Callable[[P], R]*

static teardown() → *Callable[[Callable[[P], R]], Callable[[P], R]]*

Executes decorated function's body in a global-context on-device when `device.close()` is called.

Function arguments are also set in the global context.

Can either be used as a staticmethod `@Device.teardown` for marking methods in a subclass of `Device`, or as a standard method `@device.teardown` for marking functions to a specific `Device` instance.

Parameters

- **f** (*Callable*) -- Function to decorate. Can only accept and return python literals.
- **minify** (*bool*) -- Minify cmd code prior to sending. Defaults to True.
- **register** (*bool*) -- Assign an attribute to `self.teardown` with same name as `f`. Defaults to True.
- **record** (*bool*) -- Each invocation of the executer is recorded for playback upon reconnect. Defaults to True.

terminal (*, *exit_char*='\\x1d')

Start a blocking interactive terminal over the serial port.

static thread() → *Callable*[[*P*], *R*]

static thread() → *Callable*[[*Callable*[[*P*], *R*]], *Callable*[[*P*], *R*]]

Spawn on-device thread that executes decorated function.

Can either be used as a staticmethod `@Device.thread` for marking methods in a subclass of `Device`, or as a standard method `@device.thread` for marking functions to a specific `Device` instance.

Parameters

- **f** (*Callable*) -- Function to decorate. Can only accept python literals as arguments.
- **minify** (*bool*) -- Minify cmd code prior to sending. Defaults to True.
- **register** (*bool*) -- Assign an attribute to `self.thread` with same name as `f`. Defaults to True.
- **record** (*bool*) -- Each invocation of the executer is recorded for playback upon reconnect. Defaults to True.

class Implementation(*name*: *str*, *version*: *Tuple*[*int*, *int*, *int*], *platform*: *str*, *emitters*: *Tuple*[*str*])

Implementation dataclass detailing the device.

Parameters

- **name** (*str*) -- Type of python running on device. One of {"micropython", "circuitpython"}.
- **version** (*Tuple*[*int*, *int*, *int*]) -- (major, minor, patch) Semantic versioning of device's firmware.
- **platform** (*str*) -- Board identifier. May not be consistent from MicroPython to CircuitPython. e.g. The Pi Pico is "rp2" in MicroPython, but "RP2040" in CircuitPython.
- **emitters** (*tuple*[*str*]) -- Tuple of available emitters on-device {"native", "viper"}.

emitters: *Tuple*[*str*]

name: *str*

platform: *str*

version: *Tuple*[*int*, *int*, *int*]

4.7.2 Helpers

list_devices() → List[str]

Lists available device ports.

For example:

```
['/dev/cu.usbmodem1143401', '/dev/cu.usbmodem113101']
```

Returns

Available devices identifiers.

Return type

List[str]

4.7.3 Exceptions

exception AuthenticationError

Bases: *BelayException*

Invalid password or similar.

exception BelayException

Bases: Exception

Root Belay exception class.

exception ConnectionLost

Bases: *BelayException*

Lost connection to device.

exception FeatureUnavailableError

Bases: *BelayException*

Feature unavailable for your board's implementation.

exception MaxHistoryLengthError

Bases: *BelayException*

Too many commands were given.

exception SpecialFunctionNameError

Bases: *BelayException*

Attempted to use a reserved Belay function name.

The following name rules are reserved:

- Names that start and end with double underscore, `__`.
- Names that start with `_belay` or `__belay`

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

b

`belay.exceptions`, [30](#)

INDEX

A

AuthenticationError, 30

B

belay.exceptions
 module, 30

BelayException, 30

C

close() (*Device method*), 26

ConnectionLost, 30

D

Device (*class in belay*), 26

E

emitters (*Implementation attribute*), 29

F

FeatureUnavailableError, 30

I

Implementation (*class in belay*), 29

implementation (*Device attribute*), 26

L

list_devices() (*in module belay*), 30

M

MAX_CMD_HISTORY_LEN (*Device attribute*), 26

MaxHistoryLengthError, 30

module

 belay.exceptions, 30

N

name (*Implementation attribute*), 29

P

platform (*Implementation attribute*), 29

R

reconnect() (*Device method*), 26

S

setup() (*Device static method*), 26

soft_reset() (*Device method*), 27

SpecialFunctionNameError, 30

sync() (*Device method*), 27

sync_dependencies() (*Device method*), 27

T

task() (*Device static method*), 28

teardown() (*Device static method*), 28

terminal() (*Device method*), 29

thread() (*Device static method*), 29

V

version (*Implementation attribute*), 29