Belay

Release 0.0.0

Brian Pugh

Oct 24, 2022

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 . . . 11
- 4.4 Connections . . . 12
- 4.5 How Belay Works . . . 14
- 4.6 API . . . 16

5 Indices and tables **21**

Python Module Index **23**

Index **25**

Belay is a library that enables the rapid development of projects that interact with hardware via a MicroPython or CircuitPython compatible board.

Belay works by interacting with the REPL interface of a MicroPython board from Python code running on PC.

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.

CHAPTER

ONE

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.

If you have no need to run Python code on PC, then Belay is not for you.

CHAPTER TWO

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.

CHAPTER THREE

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.

CHAPTER

FOUR

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

The `device` object has 4 important methods for projects: directly calling, `task`, `thread`, and `sync`. These are described in the subsequent subsections.

4.2.1 Call

Directly calling the `Device` instance invokes a command string on-device. For example, `device("foo = 1 + 2")` would execute the code `foo = 1 + 2` on-device. This is typically used in Belay projects to import modules and declare global variables.

4.2.2 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.

If a task is registered to multiple Belay devices, it will execute sequentially on the devices in the order that they were decorated (bottom upwards). The return value would be a list of results in order.

To explicitly call a task on just one device, it can be invoked `device.task.foo()`.

4.2.3 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.

If a thread is registered to multiple Belay devices, it will execute sequentially on the devices in the order that they were decorated (bottom upwards).

To explicitly call a thread on just one device, it can be invoked `device.thread.led_loop()`.

4.2.4 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
```

4.3 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.

4.4 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
```

(continues on next page)

(continued from previous page)

```
    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 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.

4.5.1 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.

4.5.2 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`.

4.5.3 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 form 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).

4.6 API

class Device(**args*, *startup: Optional[str] = None*, *attempts: int = 0*, ***kwargs*)

Belay interface into a micropython device.

implementation

Implementation details of device.

Type

Implementation

task(*f: Optional[Callable[..., None]] = None*, */*, *minify: bool = True*, *register: bool = True*)

Decorator that send code to device that executes when decorated function is called on-host.

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`.

thread(*f: Optional[Callable[..., None]] = None*, */*, *minify: bool = True*, *register: bool = True*)

Decorator that send code to device that spawns a thread when executed.

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`.

`__call__`(*cmd: str*, *minify: bool = True*, *stream_out: ~typing.TextIO = <_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*, *record=True*)

Execute code on-device.

Parameters

- **cmd** (*str*) -- Python code to execute.
- **minify** (*bool*) -- Minify `cmd` code prior to sending. Reduces the number of characters that need to be transmitted. Defaults to `True`.
- **record** (*bool*) -- Record the call for state-reconstruction if device is accidentally reset. Defaults to `True`.

Return type

Return value from executing code on-device.

`close`() → None

Close the connection to device.

`reconnect`(*attempts: Optional[int] = 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.

`sync`(*folder: Union[str, Path]*, *dst: str = '/'*, *keep: Union[None, list, str, bool] = None*, *ignore: Union[None, list, str] = None*, *minify: bool = True*, *mpy_cross_binary: Optional[Union[str, Path]] = 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"]`.
- **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.

exception AuthenticationError

Bases: `BelayException`

Invalid password or similar.

exception FeatureUnavailableError

Bases: `BelayException`

Feature unavailable for your board's implementation.

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

Bases: `object`

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]

exception PyboardException

Bases: `Exception`

Uncaught exception from the device.

exception SpecialFunctionNameError

Bases: `BelayException`

Reserved function name that may impact Belay functionality.

Currently limited to:

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

minify(*code: str*) → str

Minify python code.

Naive code minifying that preserves names and linenos. Performs the following:

- Removes docstrings.
- Removes comments.
- Removes unnecessary whitespace.

Parameters

code (*str*) -- Python code to minify.

Returns
Minified code.

Return type
str

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

b

belay, 17

INDEX

Symbols

__call__() (*Device method*), 17

A

AuthenticationError, 17

B

belay
    module, 17

C

close() (*Device method*), 17

D

Device (*class in belay*), 16

E

emitters (*Implementation attribute*), 18

F

FeatureUnavailableError, 18

I

Implementation (*class in belay*), 18
implementation (*Device attribute*), 16

M

minify() (*in module belay*), 18
module
    belay, 17

N

name (*Implementation attribute*), 18

P

platform (*Implementation attribute*), 18
PyboardException, 18

R

reconnect() (*Device method*), 17

S

SpecialFunctionNameError, 18
sync() (*Device method*), 17

T

task() (*Device method*), 16
thread() (*Device method*), 16

V

version (*Implementation attribute*), 18