# Belay

*Release 0.0.0*

**Brian Pugh**

**Aug 27, 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 Connections . . . 11
- 4.4 How Belay Works . . . 13
- 4.5 API . . . 14

**5 Indices and tables 17**

**Python Module Index 19**

**Index 21**

Belay is a library that enables the rapid development of projects that interact with hardware via a micropython-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.

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 Connections

Belay currently supports two connection mediums:

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

### 4.3.1 Serial

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

### 4.3.2 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")
```

## 4.4 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.

### 4.4.1 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 a few common imports on-device for convenience. Currently, these convenience imports are:

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

### 4.4.2 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`.

### 4.4.3 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).

## 4.5 API

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

Belay interface into a micropython device.

**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'>*) → Union[Generator[Union[None, bool, bytes, int, float, str, List, Dict, Set], None, None], None, bool, bytes, int, float, str, List, Dict, Set]

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`.

**Return type**
Return value from executing code on-device.

**close**()

Close the connection to device.

**sync**(*folder: Union[str, Path]*, *minify: bool = True*, *keep: Union[None, list, str] = None*, *progress_update=None*) → None

Sync a local directory to the root of 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.

**Parameters**

- **folder** (*str, Path*) -- Directory of files to sync to the root of the board's filesystem.
- **minify** (*bool*) -- Minify python files prior to syncing. Defaults to `True`.
- **keep** (*str or list*) -- Do NOT delete these file(s) on-device if not present in `folder`. Defaults to `["boot.py", "webrepl_cfg.py"]`.
- **progress** -- Partial for `rich.progress.Progress.update(task_id,...)` to update with sync status.

**exception AuthenticationError**

Bases: `Exception`

Invalid password or similar.

**exception PyboardException**

Bases: `Exception`

Uncaught exception from the device.

exception SpecialFunctionNameError

Bases: Exception

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, 15

# INDEX

## Symbols

__call__() (*Device method*), 15

## A

AuthenticationError, 15

## B

belay
    module, 15

## C

close() (*Device method*), 15

## D

Device (*class in belay*), 14

## M

minify() (*in module belay*), 16
module
    belay, 15

## P

PyboardException, 15

## S

SpecialFunctionNameError, 15
sync() (*Device method*), 15

## T

task() (*Device method*), 14
thread() (*Device method*), 15