

LIBSTP

**LEGO-INSPIRED BOTBALL SUPER
TOOLKIT FOR PROGRAMMING**

THE PROBLEM

REPEATED REINVENTION

- Each year, teams had to rewrite common functionalities (e.g., line following).
- Resulted in wasted time and duplicated effort.

INCREASING COMPLEXITY

- Robots became more advanced, but so did the codebase.
- Managing the code became:
 - Time-consuming
 - Difficult to maintain
 - Hard to understand

LACK OF STANDARDIZATION

- No unified approach across teams.
- Sharing and reusing code between teams was nearly impossible.

THE IDEA

- Two robots last year: **create3** and **wombat**.
- Initially controlled using entirely different APIs.
- Team members with limited programming experience struggled to contribute effectively.

A UNIFIED SOLUTION

- A single high-level API
- Abstracts away low-level robot details.
- Provides a unified interface for all supported robots.

GOAL

- Make the API **readable and usable** for beginners
- Still satisfy my special needs for **overengineering**




THE SOLUTION: LIBSTP

WHAT IS LIBSTP?

- Stands for "Library St. Poelten" (Or is it "Lightweight but Seriously Terrific Programming"? 🤔)
- Designed to:
 - Simplify robotics programming.
 - Enable code reuse and collaboration across teams.

KEY FEATURES

- High-Level API
- Python Supremacy 
- Beginner-Friendly (Not really, but I tried)

**TASK: DRIVE FORWARD FOR 3S,
THEN BACKWARD FOR 3S**

Here's how you can achieve this in different ways:

BAREBONE KIPR C API

```
int main()
{
    motor(0, 100); // Move forward
    motor(3, 100);
    msleep(3000);


    motor(0, -100); // Move backward
    motor(3, -100);
    msleep(3000);

    return 0;
}
```

PROS

- Simple and straightforward

CONS

-  Repetition
-  Hard to read and maintain

WITH OOP



```
left = Motor(0)
right = Motor(1)

left.set_velocity(1500) # Forward
right.set_velocity(1500)
time.sleep(3)

left.set_velocity(-1500) # Backward
right.set_velocity(-1500)
time.sleep(3)

stop_all_motors()
```

PROS

- Cleaner structure

CONS

- Still a bit repetitive 😅

LIBSTP

```
left = Motor(0)
right = Motor(1)

with TwoWheeledDevice(left, right) as device:
    device.drive_straight(for_seconds(3), Speed.Fastest)
    device.drive_straight(
        for_seconds(3),
        Speed.Fastest.backward()
    )
```

PROS

- Intuitive methods
- Less boilerplate

CONS

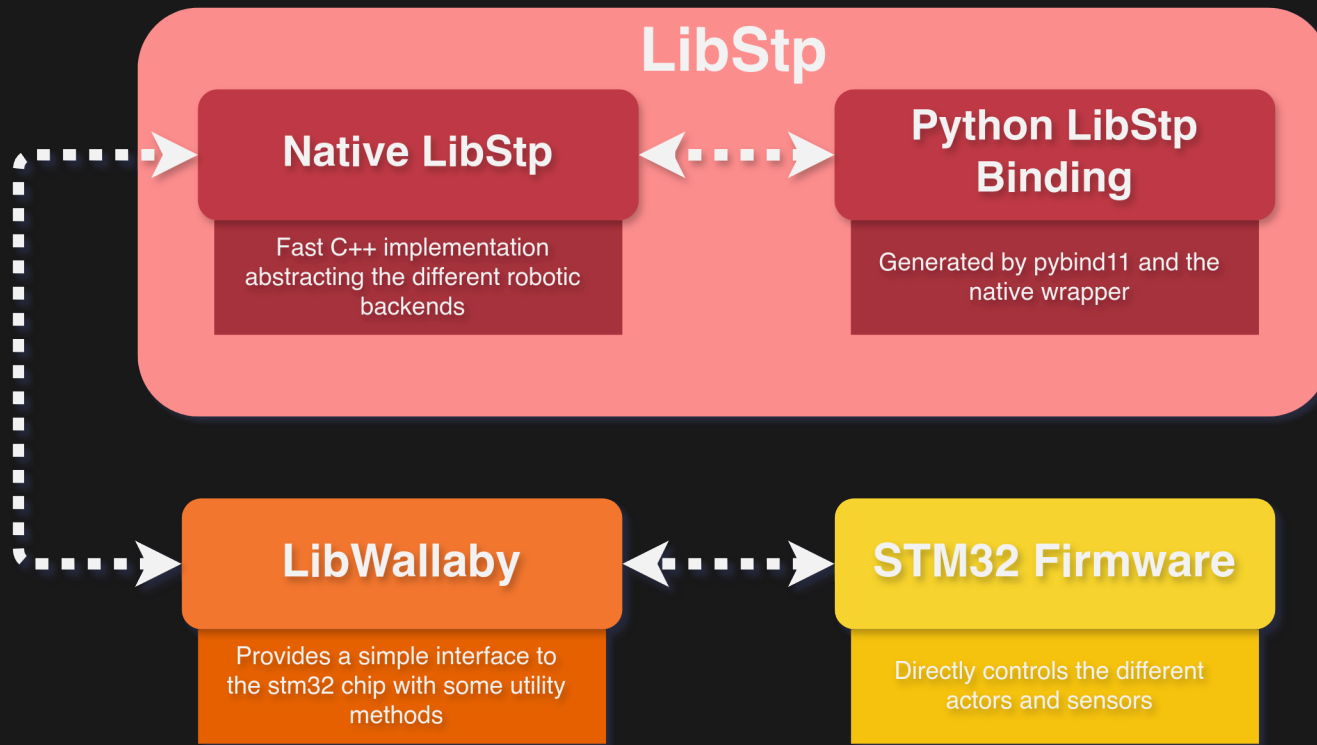
- Requires learning a new API
- Some concepts might be just an izzi bitty littly too overengineered 🧐 (You'll see)

Just as an example, the method `drive_straight` executes ~300 lines of code, including PID control, motor synchronization, and error correction. 🤯

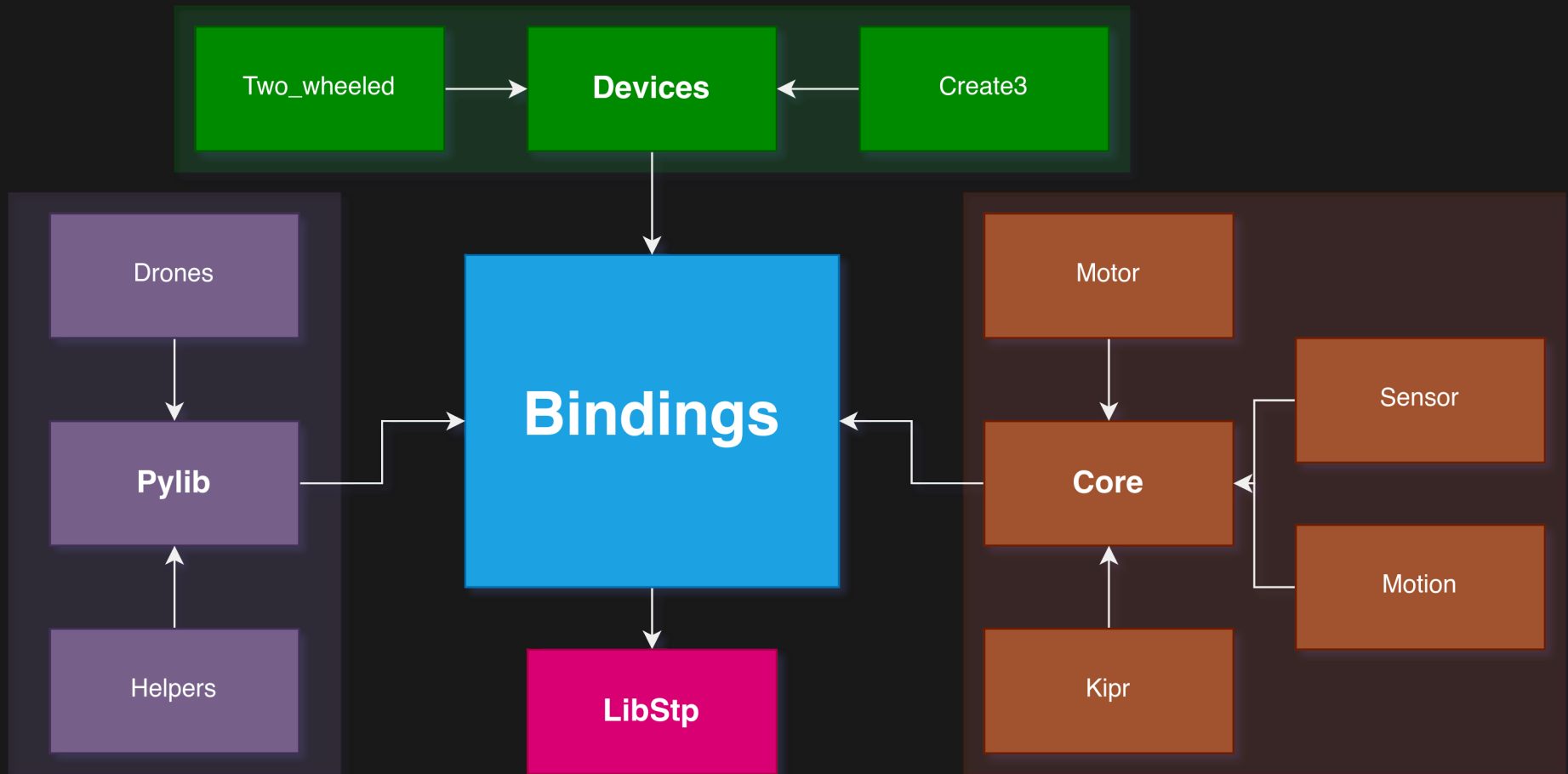
LIBSTP ARCHITECTURE

Small theoretical detour 🧐

HOW THE LIBRARY INTERFACES WITH THE ROBOT (WOMBAT)



HOW THE LIBRARY IS STRUCTURED INTERNALLY



PRACTICAL USAGE

Time to get our hands dirty! 

PREREQUISITES

The library comes preconfigured on the wombat. To write your own code, I recommend using PyCharm with an SSH Interpreter on the robot (View PyCharm docs for more info).

CREATING A ~~BACKEND~~ DEVICE

WHAT IS A DEVICE?

- A class that interfaces with the robot.
- Provides high-level methods to control the robot.
- Handles automatic calibration and more.

EXAMPLE: DEVICE IN ACTION

```
left = Motor(0)
right = Motor(1)

with TwoWheeledDevice(left, right) as robot:
    # Your code here
```


WHY USE THE `with` STATEMENT?

- Ensures proper cleanup after execution.
- Automatically handles:
 - Stopping motors.
 - Disabling servos.
 - Disposing threads.
 - Calibrating sensors.

MOST ESSENTIAL METHODS

- `drive_straight(condition, speed)`
- `rotate(condition, speed)`

CONDITIONS

Define when to stop executing a command.

UNDEFINED CONDITIONS

Important: Lerp can't be used with them

- `while_true(lambda: x <= 5)` - Run until the lambda returns False
- `while_false(lambda: x <= 5)` - Run until the lambda returns True

DEFINED CONDITIONS

- `for_seconds(seconds)` - Time-based condition
- `for_distance(distance_cm)` - Distance-based condition
- `for_ccw_rotation(degrees)` - CCW Rotation-based condition
- `for_cw_rotation(degrees)` - CW Rotation-based condition

SPEEDS

The library provides five speed levels:

- `Speed.Slowest`
- `Speed.Slow`
- `Speed.Medium`
- `Speed.Fast`
- `Speed.Fastest`

Note: Speeds are forward by default. To move backward, use the `backward()` method.

MANUALLY CONFIGURING SPEEDS

Speed allows you to configure speeds manually:

```
my_speed = Speed(  
    forward_percent=0.1, # 10% of max speed (~20 cm/s for tv  
    angular_percent=0.5 # 50% of max angular speed (~2.3 rad  
)
```

Note: `drive_straight` as example only use `forward_percent`.

SPEEDFUNCTION

`drive_straight` as example accepts a constant speed or a function that returns a speed:

```
robot.drive_straight(  
    for_distance(10),  
    lerp(Speed.Fastest, Speed.Slowest)  
)
```

This makes the robot go from `Speed.Fastest` to `Speed.Slowest` over the course of the distance.

**BASICS OVER, LET'S
TALK ABOUT SOME
OVERENGINEERING**



PROPER ARCHITECTURE

I found that it's best to separate your code into different modules:

- `main.py` - Entrypoint & Main Module
- `setup.py` - Module for pre-run setup
- `modules/` - Additional modules for specific tasks
- `definitions/` - Special folder housing definitions

MODULES

A module is:

- A python class that inherits from Module.
- Modules help keep the separation of concerns principle.
- Modules ideally call other high level methods.

DEFINITIONS

There is a base definition. It's supposed to house all servo, motor, and sensor references.

A definition can be easily modified for specific tables with definition overrides.

EXAMPLE

Every concept is best explained with an example.

Look at the example code in the `examples` folder
(Ideally the `examples/robot/gcer2024-
create3` folder).

Note: The example code still uses the old api, but it should give you a good idea of how to structure your code.

EXAMPLE CODE SEGMENT

```
class Get5Habitats(Create3Module):
    def __init__(self, robot: RobotBackend,
definitions: Definitions):
        super().__init__("get_5_habitats",
            robot, definitions)

    def run(self):
        self.robot.drive_straight(for_seconds(0.3),
            constant(Speed.Slowest))
        self.robot.rotate(for_cw_rotation(45),
            constant(Speed.Fastest))
```

SENSORS

Many different sensor implementations are available:

- `AnalogSensor(port)`
- `DigitalSensor(port)` - `is_clicked`
- `DistanceSensor(port)` - `get_distance` (in cm)
- `LightSensor(port)` - `is_on_black`,
`wait_for_light`
- IMUSensors (gyro, accel, mag, etc)

SENSOR FILTERING

Multiple filters have been implemented. They try to reduce noise and make the sensor readings more reliable.

- AvgFilter
- FirstOrderLowPassFilter
- MovingAverageFilter
- ...

SENSOR USAGE

```
gyro_x = GyroXSensor(FirstOrderLowPassFilter(0.7))
gyro_y = GyroYSensor(FirstOrderLowPassFilter(0.7))
gyro_z = GyroZSensor(FirstOrderLowPassFilter(0.7))

wait_for_button_click()
while not is_button_clicked():
    x = gyro_x.get_value()
    y = gyro_y.get_value()
    z = gyro_z.get_value()
    print(f'X: {x}, Y: {y}, Z: {z}')
    time.sleep(0.1)
```

ACTUATORS

Actuators are used to control the robot's movement.

MOTORS

```
Motor(port, reverse_polarity)
```

MOTOR FUNCTIONS

- `set_velocity(velocity)` - -1500 - 1500
- `stop()` - Active braking
- `get_current_position_estimate()` - BackEMF
- ... (Docs)

SERVOS

Servo(port)

SERVO FUNCTIONS

- `set_position(position)` - 0 - 2047
- `enable()` - Enable the servo
- `disable()` - Disable the servo
- `slowly_set_position(position, speed, interpolation)` - Slowly set the servo position
- ... (Docs)

SERVO LIKE MOTOR

```
ServoLikeMotor(port,  
reverse_polarity)
```

This is a motor that has the same functionality as a servo, but less accurate

THANK YOU!