j5 Documentation

Release 0.10.0

j5 Contributors

Jan 07, 2021

Contents:

CHAPTER 1

What is j5?

j5 is a Python 3 framework that aims to make building consistent APIs for robotics easier. It was created to reduce the replication of effort into developing the separate, yet very similar APIs for several robotics competitions. Combining the common elements into a single library with support for various hardware gives a consistent feel for students and volunteers. This means more time to work on building robots!

Please note that this documentation is not aimed at the average competitor. It is for used by developers of the API, competition volunteers and more advanced students wishing to extend on our API for their robots. Support is discretionary to the individual competition. *j5* will not provide direct support to compeititors at the time of writing.

1.1 Installation

j5 is really easy to install.

You will need the following installed on your machine:

- Python 3.6 or higher
- python3-pip (for package management)
- pipenv (optional)

1.1.1 pipenv (recommended)

The recommended installation method is to use pipenv, an excellent tool that combines a package manager with virtual environments.

Simply run: `pipenv install j5`

If you want Zoloto CV support: `pipenv install j5[zoloto-vision]`

You can now import *j5* into your libraries. Awesome!

1.1.2 pip

You can use pip to install *j5*. You will either need to install it system-wide or manage a virtual environment.

Simply run: `pip install j5`

You can now import *j5* into your libraries. Awesome!

1.2 Quick Start Guide

Firstly, you will need to ensure that you have installed j5. You will also need a working knowledge of Python 3.

1.2.1 Your First Robot

The recommended way to use *j5* is to first define what the structure of your robot looks like.

You will probably want

```
from j5 import BaseRobot


class MyRobot(BaseRobot):
    """My Basic Robot definition."""


r = MyRobot()
```

1.2.2 Adding Boards

To give you robot some functionality, you will need to define what boards are available on your robot.

```
from j5 import BaseRobot, BoardGroup
from j5.backends.console.sr.v4 import (
    SRV4MotorBoardConsoleBackend,
    SRV4PowerBoardConsoleBackend,
)
from j5.boards.sr.v4 import MotorBoard, PowerBoard


class MyRobot(BaseRobot):
    """A robot with a few boards."""

    def __init__(self) -> None:
        self._power_boards = BoardGroup.get_board_group(
            PowerBoard,
            SRV4PowerBoardConsoleBackend,
        )
        self.power_board = self._power_boards.singular()  # Restrict to exactly one␣
↪board.

        self.motor_boards = BoardGroup.get_board_group(
            MotorBoard,
            SRV4MotorBoardConsoleBackend,
        )
```

(continues on next page)

(continued from previous page)

```
r = MyRobot()

print(f"Found Power Board: {r.power_board.serial_number}")
print(f"Power Board Firmware: {r.power_board.firmware_version}")

# Access a board specific function
r.power_board.wait_for_start_flash()

print(f"Found {len(r.motor_boards)} Motor Board(s):")

# Iterate over the boards in a board group
for board in r.motor_boards:
    print(f" - {board.serial_number} - Version {board.firmware_version}")

# Access board by serial number
r.motor_boards["218312"].make_safe()
```

In order to add some boards to your robot, you will need to define the `BoardGroup` for your board. A `BoardGroup` is a group of boards attached to your robot. A `BoardGroup` can contain 0 or more of the specified board. You can also call `singular()` on your `BoardGroup`, and it will throw an error if there is not exactly one board of that type connected.

If your robot does not consistent of a modular kit, and is entirely contained within one unit, you do not have to use the board separation, you can instead directly expose components to the use.

Note that whilst we can iterate over a `BoardGroup` and access a board in a `BoardGroup` by serial, we cannot access a board using array notation.

1.2.3 Using Components

Whilst it is useful to be able to access attributes and functions that are specific to a board, the real power of *j5* is found when you access components and functionality on those boards. *j5* has defined a consistent interface for those components, even if they are on separate devices.

```
from j5 import BaseRobot, BoardGroup
from j5.backends.console.sr.v4 import SRV4PowerBoardConsoleBackend
from j5.boards.sr.v4 import PowerBoard


class MyRobot(BaseRobot):
    """A robot with a few boards."""

    def __init__(self) -> None:
        self._power_boards = BoardGroup.get_board_group(
            PowerBoard,
            SRV4PowerBoardConsoleBackend,
        )
        self.power_board = self._power_boards.singular()  # Restrict to exactly one␣
↪board.

        # Expose just a component to the user.
        self.big_led = self.power_board.outputs[0]
```

(continues on next page)

(continued from previous page)

```
r = MyRobot()

# Ensure all outputs on the power board are off.

for output in r.power_board.outputs:
    output.is_enabled = False

# Turn on the big LED
r.big_led.is_enabled = True
```

The usual method to access components is to use the definition on the board. It is also possible to expose a component, or even a single attribute on a component as a top level attribute of your Robot object.

1.3 Concepts

1.3.1 Abstractions

j5 utilises a number of abstractions to enable similar APIs across platforms and hardware. This page explains design decisions behind the major abstractions and how to use them correctly.

Component

A component is the smallest logical part of some hardware.

A component will have the same basic functionality no matter what hardware it is on. For example, an LED is still an LED, no matter whether it is on an Arduino, or the control panel of a jumbo jet; it still can be turned on and off.

The component should expose a user-friendly API, attempting to be consistent with other components where possible.

Validation of user input should be done in the component.

Implementation

A component is implemented by sub-classing the `j5.components.Component`.

It is uniquely identified on a particular *`j5.boards.Board`* by an integer, which is usually passed into the constructor.

Every instance of a component should have a reference to a *`j5.backends.Backend`*, that implements the relevant `j5.components.Interface`.

The relevant `j5.components.Interface` should also be defined.

```
class LED(Component):
    """A standard Light Emitting Diode."""

    def __init__(self, identifier: int, backend: LEDInterface) -> None:
        self._backend = backend
        self._identifier = identifier

    @staticmethod
    def interface_class() -> Type[LEDInterface]:
```

(continues on next page)

(continued from previous page)

```
        """Get the interface class that is required to use this component."""
        return LEDInterface

    @property
    def identifier(self) -> int:
        """An integer to identify the component on a board."""
        return self._identifier

    @property
    def state(self) -> bool:
        """Get the current state of the LED."""
        return self._backend.get_led_state(self._identifier)

    @state.setter
    def state(self, new_state: bool) -> None:
        """Set the state of the LED."""
        self._backend.set_led_state(self._identifier, new_state)
```

Interface

An interface defines the low-level methods that are required to control a given component.

Implementation

An interface should sub-class `j5.components.Interface`.

The interface class should contain abstract methods required to control the component.

```
class LEDInterface(Interface):
    """An interface containing the methods required to control an LED."""

    @abstractmethod
    def get_led_state(self, identifier: int) -> bool:
        """Get the state of an LED."""
        raise NotImplementedError  # pragma: no cover

    @abstractmethod
    def set_led_state(self, identifier: int, state: bool) -> None:
        """Set the state of an LED."""
        raise NotImplementedError  # pragma: no cover
```

Board

A Board is a class that exposes a group of components, used to represent a physical board in a robotics kit.

The Board class should not directly interact with any hardware, instead making calls to the Backend class where necessary, and preferably diverting interaction through the component classes where possible.

Implementation

An interface should sub-class *j5.boards.Board*.

It will need to implement a number of abstract functions on that class.

Components should be created in the constructor, and should be made available to the user through properties. Care should be taken to ensure that users cannot accidentally override components.

A backend should also be passed to the board in the constructor, usually done in *j5.backends.Backend.discover()*

A notable method that should be implemented is *j5.boards.Board.make_safe()*, which should call the appropriate methods on the components to ensure that the board is safe in the event of something going wrong.

```
if TYPE_CHECKING:  # pragma: no cover
    from j5.components import Component  # noqa: F401


class MotorBoard(Board):
    """Student Robotics v4 Motor Board."""

    name: str = "Student Robotics v4 Motor Board"

    def __init__(
            self,
            serial: str,
            backend: Backend,
            *,
            safe_state: MotorState = MotorSpecialState.BRAKE,
    ):
        self._serial = serial
        self._backend = backend
        self._safe_state = safe_state

        self._outputs = ImmutableList[Motor](
            Motor(output, cast(MotorInterface, self._backend))
            for output in range(0, 2)
        )

    @property
    def serial_number(self) -> str:
        """Get the serial number."""
        return self._serial

    @property
    def firmware_version(self) -> Optional[str]:
        """Get the firmware version of the board."""
        return self._backend.firmware_version

    @property
    def motors(self) -> ImmutableList[Motor]:
        """Get the motors on this board."""
        return self._outputs
```

Backend

A backend implements all of the interfaces required to control a board.

A backend also contains a method that can discover boards.

Multiple backends can be implemented for one board, but a backend can only support one board. This could be used for implementing a simulated version of a board, in addition to the hardware implementation.

Backends can also validate is data is suitable for them, and throw an error if not; for example *j5.backends.hardware.env.NotSupportedByHardwareError*.

Implementation

```python
class SRV4MotorBoardConsoleBackend(
    MotorInterface,
    Backend,
):
    """The console implementation of the SR v4 motor board."""

    board = MotorBoard

    @classmethod
    def discover(cls) -> Set[Board]:
        """Discover boards that this backend can control."""
        return {cast(Board, MotorBoard("SERIAL", cls("SERIAL")))}

    def __init__(self, serial: str, console_class: Type[Console] = Console) -> None:
        self._serial = serial

        # Initialise our stored values for the state.
        self._state: List[MotorState] = [
            MotorSpecialState.BRAKE
            for _ in range(0, 2)
        ]

        # Setup console helper
        self._console = console_class(f"{self.board.__name__}({self._serial})")

    @property
    def serial(self) -> str:
        """The serial number reported by the board."""
        return self._serial

    @property
    def firmware_version(self) -> Optional[str]:
        """The firmware version reported by the board."""
        return None  # Console, so no firmware

    def get_motor_state(self, identifier: int) -> MotorState:
        """Get the current motor state."""
        # We are unable to read the state from the motor board, in hardware
        # so instead of asking, we'll get the last set value.
        return self._state[identifier]

    def set_motor_state(self, identifier: int, power: MotorState) -> None:
        """Set the state of a motor."""
        if identifier not in range(0, 2):
            raise ValueError(
                f"Invalid motor identifier: {identifier}, valid values are: 0, 1",
            )
        self._state[identifier] = power
        if isinstance(power, MotorSpecialState):
            power_human_name = power.name
        else:
```

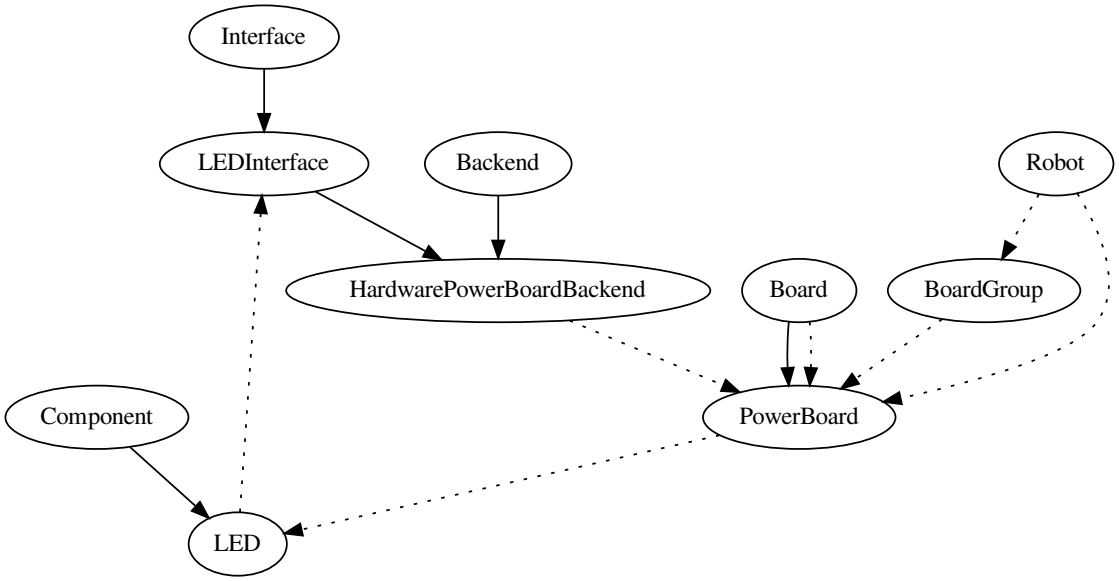
(continues on next page)

(continued from previous page)

```
52              power_human_name = str(power)
53          self._console.info(f"Setting motor {identifier} to {power_human_name}.")
```

Class Diagram

The below diagram shows a class having instances of another class as an attribute with a dotted line. Solid lines indicate that there is a sub-class relationship



1.3.2 Comparison to alternatives

Similar Libraries

j5 was designed to supersede a number of similar libraries. The table below gives a brief comparison between *j5*, *robot-api* / *robotd* and *sr.robot*.

Feature	j5	robot-api / robotd	sr.robot
Cross-Platform Support	Yes	No (Requires Linux + udev + systemd)	No (Requires Linux + udev)
Custom / Game Logic without core changes	N/A	No	No
Developer Documentation	Yes	No	No
Explanative error messages	Yes	No (Pipe Error)	Mostly
Advanced Fiducial Marker Support	Yes (Zoloto)	Partial (sb-vision)	Yes (Libkoki)
OSI Licence	Yes	Yes	No
PEP8 Compliant	Yes	Non-strict	No
PyPI	Yes	No	No
Python 3	Yes	Yes	No
Run code without hardware	Yes (ConsoleEnvironment)	No	No
Supports multiple environments / backends	Yes	Yes	No
Supports SourceBots Servo Board	Partial	Yes	No
Supports SR v4 Kit	Yes	Partial Support	Yes
Test Coverage	> 98%	Some	No
Type Checking	Yes	Partial	No
User Documentation	N / A	Yes	Yes
Versioning	Yes (SemVer)	Yes	No

Robot Operating System (ROS)

> The Robot Operating System (ROS) is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms.

The brief paragraph above makes it sound like ROS is very similar to *j5* and the basic idea behind it is. However, *j5* is more suitable for students due to the following:

- Hardware implementation is Python, easier to understand / debug than C++.
- Standard libraries can be used in student code to add custom hardware in *j5*, i.e from Adafruit.
- Smaller codebase.
- Simpler architecture.
- ROS is a real-time operating system, which presents a different way of programming than most students will have been taught.
- ROS is aimed at research environments, *j5* is aimed specifically for robotics competitions.
- ROS is complex - The ROS framework is a multi-server distributed computing environment allowing software applications to communicate across server boundaries and thereby acting as one software system. - We do not need distributed computing. - The more complicated the system, the harder it is to debug. We want to allow students to debug their code.
- ROS does not expose a common API for various hardware. Instead, the appropriate messages must be published to that hardware, which will be different.
- ROS does not have a security model.

- ROS has no automated system for upgrading firmware, nor for updating itself.
- ROS has no configuration management system.
- The ROS messaging system has a fairly large overhead.
- It is non-trivial to add extra hardware support in ROS, raising the barrier to students using non-provided components.

1.3.3 Philosophy Behind j5

Some Background

Student Robotics is a charity that was originally founded by a group of students at the University of Southampton with the goal of bringing the excitement of engineering and the challenge of coding to young people through robotics. This has involved running an annual robotics competition almost every year since 2008, where groups of sixth form students are given some robotics kit, time and mentoring to develop a competitive robot for a unique challenge. In order to reduce the barrier to entry for the competition, it is essential that a knowledge of low level programming and hardware is not required by the students. Thus, a Python API is usually supplied alongside hardware that is developed in order to make things easier.

In 2017 / 18, Student Robotics underwent some restructuring and as a result did not hold a competition. To meet the demand of teachers for the competition, volunteers created two independent competitions SourceBots and Robocon. Both competitions designed and built their own robotics kits that were very similar to the current Student Robotics kit, yet completely incompatible with both each other and the previous kit.

Unification

Following the reappearance of Student Robotics to the scene in late 2018, there were now three separate, very similar, and also incompatible robotics kits that were being used for the same purpose. None of the kits were perfect, and volunteers didn't want to replicate the effort three times for everything. Thus it makes sense to combine the joint efforts of all three teams of kit developers into one. This is the goal of *j5*. *j5* is a single library that provides a uniform interface and API to students for all three kits. Whilst code will not be directly portable between the kits, it will also not be very hard to port code between them. As a library, *j5* still allows the development teams at the individual competitions to have some degree of customisation over how their kit is used.

Goals

There are some goals behind the *j5* project:

- To be compatible with a variety of relevant current and future robotics kits.
- To use the latest stable version of software and be continuously maintained, even between and during competitions.
- To be an example to students of what good code should look like.
- To unify all existing robotics kits and simulators into one codebase.
- Open by default, no hidden documentation, features or meetings.

1.4 Supported Hardware

j5 is designed to support hardware from many different competitions, some of which are officially supported by the project.

Note: Currently, these are integrated into the main `j5` module, but in the future they are due to be separated separate python modules and uploaded to PyPI separately. This ensures that the core of *j5* stays small.

1.4.1 SourceBots

The SourceBots Arduino firmware is supported.

When it is developed, `j5` will support the new computer vision board that SourceBots are developing.

1.4.2 Student Robotics

Note: Please note that Student Robotics does not officially endorse *j5*, although we hope they will in the future.

Student Robotics challenges teams of 16 to 18 year-olds to design, build and develop autonomous robots to compete in their annual competition. After announcing the year's game, they give teams six months to engineer their creations. They mentor teams throughout this time, as well as supply them with a kit which provides a framework they can build their robot around.

Student Robotics is currently on it's fourth generation of robotics kit, which is mostly based around the ODROID U3 and some custom designed hardware that's based on STM32 microcontrollers. The kit communicates with the ODROID using USB, which has proven to be a more reliable communication method than their previous kits.

`j5` supports the following Student Robotics Hardware:

- Power Board v4
- Motor Board v4
- Servo Board v4

Support for the Ruggeduino is planned.

1.5 Development

This section is about development of *j5*.

1.5.1 Getting Started

j5 is developed on GitHub and pull requests should be submitted there. If you have write access to the repository, you optionally can develop your changes on a branch within the main repository. Alternatively, please fork the *j5* repository and pull request from there.

If you are working on something that has an existing issue open on the *j5* repository, please ensure that you assign the issue to yourself such that duplication of work does not accidentally occur.

If you need help with Git, there are some good tutorial resources here:

- Git - The Simple Guide
- GitHub - Learning Git
- Atlassian Git Tutorial

Setting Up

You will need the following installed on your machine:

- Python 3.6 or higher
- python3-pip (for package management)
- GNU Make
- poetry

Now clone the repository from GitHub into a folder on your local machine.

Inside that folder, we need to tell *poetry* to install the dev dependencies: `poetry install`

You can now enter the virtual environment using `poetry shell` and develop using your IDE of choice.

Testing

As our code is used and viewed by students, we have a high standard of code within *j5*. All code must be statically typed, linted and covered in unit tests.

You can run all of the required tests with one command: `make`.

Unit Testing

We use *pytest* and *coverage.py* to do our unit testing.

Execute the test suite: `make test`

If you wish to view the *HTML* output from *coverage.py* to help you find statements that are not covered by unit tests, you can run the test suite in *html-cov* mode.

Execute the test suite in *html-cov* mode: `make test-cov`

Linting

We use *flake8* and a number of extensions to ensure that our code meets the *PEP 8* standards.

Execute the linter: `make lint`

Static Type Checking

We use *mypy* to statically type check our code.

Execute Type Checking: `make type`

Documentation

We are using *Sphinx* to generate documentation for the project.

All documentation can be found in the `docs/` folder.

Generate HTML Documentation: `make html`

1.5.2 Communications

Most of the communications for *j5* occur on GitHub, but there are a few other comms channels that we also make use of. This page explains what we use each platform for.

GitHub

GitHub is a code hosting and project collaboration platform. We use it to track issues and changes for the project, in addition to hosting our code repositories.

We have a GitHub organisation which groups all of our repositories together.

Mailing List

The mailing list is used for communications that need to be properly discussed, such as major changes to the API, or a change in the project specifications.

Any meetings that occur are also announced on the mailing list.

Our mailing list is hosted on Google Groups. You need to be a member of the list to post to it.

Slack

We also have casual discussion in `#j5` on the SRO Slack.

You can automatically join with a `@soton.ac.uk` email address. If you are not a member of the University of Southampton, please ask for an invite via GitHub or the mailing list.

Meetings

We will have infrequent meetings to discuss the state of the project. These will be announced via the mailing list.

There is usually a canonical physical location, but most will join using Google Meet. Any details of meetings will be shared in the announcement email.

1.5.3 Releases

This page contains information on how we make releases of *j5* and what the process for releasing is.

Milestones

Every version that will be released, with the exception of hotfix releases, will be added as a milestone on GitHub, such that one can see at a glance what work needs to be done before those features are released. All issues with the exception of patches are likely to be added to a milestone so that we know when it will be released to end users.

Locations

We release *j5* to two major locations:

- PyPI
- GitHub

The most important of these two locations is PyPI, as this will allow users to specify *j5* as a dependency for their API and *pip* will be able to resolve and download our package, and our dependencies also.

The release should also be created as a 'release' on GitHub, with a git tag, version number and description of the changes that have been made since the previous release. Any binary files associated with the release, such as wheels, should also be uploaded to GitHub at this point. It should be ensured that these binaries match those that are uploaded to PyPI.

Version Strategy

As a general rule, *j5* will follow Semantic Versioning.

Early Development

During early development of *j5*, we will be using version numbers of the format *0.y.z*, where *y* increments when new features are added and *z* increments when a patch version is released. During this phase of development, the API is considered to be unstable and subject to change.

Mature Development

After early development is finished, *j5* will use a combination of Semantic Versioning and ideas taken from Git Flow. In particular, the concept of release branches for major versions and having multiple major and minor versions in maintenance at any one time. Those who are running a competition should never ship an increment to the major version number during a competition cycle as this will break the code of their teams. All versions where the major version number is greater than *0* should be considered to be stable and will undergo additional testing before release.

Release Process

- Make a commit that bumps the version numbers in *j5/__init__.py* and *pyproject.toml* to the new version, and merge it to master.
- Ensure you have this commit checked out, then run *poetry publish –build* and follow the instructions. This publishes the release to PyPI.
- Go to https://github.com/j5api/j5/releases/new.
- Tag version should be of the form "v0.7.3".
- Title should be of the form "Release 0.7.3".
- Enter a release description outlining the changes made since the previous release. *git log v0.7.2..master* might be useful here.
- Upload the artifacts that were built by *poetry publish –build* earlier. They can be found in the *dist* subdirectory.
- Click publish!

1.5.4 Submitting Changes

This page details how to make and submit changes to the *j5* codebase.

Repositories

The main repository for *j5* is available on GitHub, and Pull Requests should be submitted there.

There is an additional repository available on GitLab, although this is a mirror of the GitHub repo, and changes should not be submitted there.

Discussing Changes

In many cases, changes should be discussed on an issue prior to beginning work on them, particularly where the changes make a breaking change to external APIs or are otherwise significant.

At the discussion stage, other contributors can give early feedback on the idea and suggest ways to implement it.

Publishing Changes

The first step to submit changes is to publish them so that other contributors can review and give feedback.

Branches should be published on GitHub to a fork of the j5 repository.

Pull Requests

The next stage is to submit a pull request (PR) to the GitHub repository. The PR will be used to give feedback on, and discuss the changes with the contributor.

When making your PR, consider the following:

- Why do you want to make these changes?
- Which parts of the codebase do your changes affect?
- Have you updated the relevant documentation?
- Do your changes break the external API? Add the `semver-major`, `semver-minor` or `semver-patch` label as appropriate.
- Do you want multiple reviewers to approve your code before merging?

If there is a related issue, make sure that your reference the issue number in your PR.

You may optionally request specific reviewers, and GitHub will often suggest people.

Review

Once changes have been submitted, it enters the review stage.

Guidelines for Contributors

The first part of review is automated. CircleCI will automatically check your code. You can click on the green tick, or red cross to see more information once the tests have been run. Your code cannot be merged until the automated tests have passed.

You should receive feedback on your code from reviewers. You can then discuss the feedback, and make changes as needed. When a reviewer is satisfied with your code, it will receive an approval and will be merged. You may find that several cycles of review and changes are needed until your code is ready to be merged.

Guidelines for Reviewers

When reviewing, ensure that you consider the following:

- Most importantly, give *positive* feedback. Our contributors dedicate their time and energy to submitting changes and

we need to ensure that we appreciated that. * Check the results of the CI. Has it failed? Consider suggesting to the contributor why? * Where possible, use the "Suggest Changes" feature on GitHub, this makes it easy to show what you are suggesting, and allows the contributor to instantly apply your suggestions. * In general, if you think many changes will be needed, focus on the major changes in the first round of review. * Check any referenced issues to ensure that you have context for the changes.

Merge

PRs can be merged once there is an approval. Code would usually be merged by the approving reviewer. However, there are some circumstances where this may not be desired:

- Contributor has requested multiple review approvals
- Changes would be breaking and we cannot release a major version currently. This issue will be mitigated with LTS

branches, but we do not have any of these at this time.

1.6 j5

1.6.1 j5 package

Subpackages

j5.backends package

Subpackages

j5.backends.console package

Subpackages

j5.backends.console.sb package

Submodules

j5.backends.console.sb.arduino module

Module contents

Backends for SourceBots boards in the Console Environment.

j5.backends.console.sr package

Subpackages

j5.backends.console.sr.v4 package

Submodules

j5.backends.console.sr.v4.motor_board module

j5.backends.console.sr.v4.power_board module

j5.backends.console.sr.v4.servo_board module

Module contents

Module contents

Backends for Student Robotics boards in the console environment.

j5.backends.console.zoloto package

Submodules

j5.backends.console.zoloto.camera_board module

Module contents

Backends for the Zoloto computer vision system.

Submodules

j5.backends.console.console module

Console helper classes.

class j5.backends.console.console.**Console**(*descriptor: str, print_function: Callable = <built-in function print>, input_function: Callable = <built-in function input>*)

Bases: object

A helper class for console backends.

info(*message: str*) → None
Print information to the user.

read(*prompt: str, return_type: Optional[Type[T]] = <class 'str'>*) → T
Get a value of type 'return_type' from the user.

Module contents

Backends for the Console Environment.

class j5.backends.console.**Console**(*descriptor: str, print_function: Callable = <built-in function print>, input_function: Callable = <built-in function input>*)

Bases: object

A helper class for console backends.

info(*message: str*) → None
Print information to the user.

read(*prompt: str, return_type: Optional[Type[T]] = <class 'str'>*) → T
Get a value of type 'return_type' from the user.

j5.backends.hardware package

Subpackages

j5.backends.hardware.j5 package

Submodules

j5.backends.hardware.j5.arduino module

j5.backends.hardware.j5.raw_usb module

j5.backends.hardware.j5.serial module

Module contents

Abstract hardware backend implementions provided by j5.

j5.backends.hardware.sb package

Submodules

j5.backends.hardware.sb.arduino module

Module contents

Backends for SourceBots boards in the Hardware Environment.

j5.backends.hardware.sr package

Subpackages

j5.backends.hardware.sr.v4 package

Submodules

j5.backends.hardware.sr.v4.motor_board module

j5.backends.hardware.sr.v4.power_board module

j5.backends.hardware.sr.v4.ruggeduino module

j5.backends.hardware.sr.v4.servo_board module

Module contents

Module contents

Backends for Student Robotics boards in the hardware environment.

j5.backends.hardware.zoloto package

Submodules

j5.backends.hardware.zoloto.camera_board module

Module contents

Submodules

j5.backends.hardware.env module

The hardware Environment.

exception j5.backends.hardware.env.**NotSupportedByHardwareError**
Bases: Exception

The hardware does not support that functionality.

Module contents

Backends for the hardware environment.

exception j5.backends.hardware.**NotSupportedByHardwareError**
Bases: Exception

The hardware does not support that functionality.

Submodules

j5.backends.backend module

The base classes for backends.

class j5.backends.backend.**Backend**
Bases: object

The base class for a backend.

A backend is an implementation of a specific board for an environment.

It can hold data about the actual board it is controlling. There should be a ratio of one instance of a Backend to one instance of a Board. The Backend object should not hold any references to the Board, instead having it's methods executed by the code for the individual Board.

A Backend usually also implements a number of ComponentInterfaces which thus allow a physical component to be controlled by the abstract Component representation.

board
Type of board this backend implements.

classmethod discover() → Set[Board]
Discover boards that this backend can control.

firmware_version
The firmware version of the board.

class j5.backends.backend.**BackendMeta**
Bases: abc.ABCMeta

The metaclass for a backend.

Ensures that the backend implements the correct interfaces when instantiated. It does this by checking that the class inherits from the interface classes defined on the Components in backend.board.supported_components.

exception j5.backends.backend.**CommunicationError**
Bases: Exception

A communication error occurred.

This error is thrown when there is an error communicating with a board, if a more specific exception is available, then that may be thrown instead, but it should inherit from this one.

j5.backends.environment module

Environment class and related functions.

class j5.backends.environment.**Environment**(*name: str*)
Bases: object

A collection of backends that can work together.

A number of Backends that we wish to use in a grouping, such as those that all work together in hardware can be added to this group. We can then pass Environments to a Robot object, so that the Robot object can call different methods based on where it is being used.

e.g Hardware Environment

We have *n* boards of *n* different types. They are all physical hardware. We create an Environment containing the Backends to control the physical hardware for our boards.

It is later realised that we want to test code without the physical hardware. We can add Console backends to an environment, and instantiate our Robot object with that environment, so that the console is manipulated rather than the hardware.

This allows for a high degree of code reuse and ensures API compatibility in different situations.

get_backend(*board: Type[Board]*) → Type[j5.backends.backend.Backend]
Get the backend for a board.

get_board_group(*board: Type[BoardT]*) → j5.boards.board_group.BoardGroup[~BoardT, j5.backends.backend.Backend][BoardT, j5.backends.backend.Backend]
Get a board group for the given board type.

merge(*other: j5.backends.environment.Environment*) → None
Merge in the board-backend mappings from another environment.

This allows vendors to predefine Environments and API authors can then merge several vendor environments to get the one that they need for their API.

This method will fail if any board is defined in both environments, as it is unclear which one has the correct mapping.

register_backend(*backend: Type[j5.backends.backend.Backend]*) → None
Register a new backend with this environment.

supported_boards
The boards that are supported by this environment.

Module contents

Backend classes.

class j5.backends.**Backend**
Bases: object

The base class for a backend.

A backend is an implementation of a specific board for an environment.

It can hold data about the actual board it is controlling. There should be a ratio of one instance of a Backend to one instance of a Board. The Backend object should not hold any references to the Board, instead having it's methods executed by the code for the individual Board.

A Backend usually also implements a number of ComponentInterfaces which thus allow a physical component to be controlled by the abstract Component representation.

board
Type of board this backend implements.

classmethod discover() → Set[Board]
Discover boards that this backend can control.

firmware_version
The firmware version of the board.

class j5.backends.**BackendMeta**
Bases: abc.ABCMeta

The metaclass for a backend.

Ensures that the backend implements the correct interfaces when instantiated. It does this by checking that the class inherits from the interface classes defined on the Components in backend.board.supported_components.

exception j5.backends.**CommunicationError**
Bases: Exception

A communication error occurred.

This error is thrown when there is an error communicating with a board, if a more specific exception is available, then that may be thrown instead, but it should inherit from this one.

class j5.backends.**Environment**(*name: str*)
Bases: object

A collection of backends that can work together.

A number of Backends that we wish to use in a grouping, such as those that all work together in hardware can be added to this group. We can then pass Environments to a Robot object, so that the Robot object can call different methods based on where it is being used.

e.g Hardware Environment

We have *n* boards of *n* different types. They are all physical hardware. We create an Environment containing the Backends to control the physical hardware for our boards.

It is later realised that we want to test code without the physical hardware. We can add Console backends to an environment, and instantiate our Robot object with that environment, so that the console is manipulated rather than the hardware.

This allows for a high degree of code reuse and ensures API compatibility in different situations.

get_backend(*board: Type[Board]*) → Type[j5.backends.backend.Backend]
Get the backend for a board.

get_board_group(*board: Type[BoardT]*) → j5.boards.board_group.BoardGroup[~BoardT, j5.backends.backend.Backend][BoardT, j5.backends.backend.Backend]
Get a board group for the given board type.

merge(*other: j5.backends.environment.Environment*) → None
Merge in the board-backend mappings from another environment.

This allows vendors to predefine Environments and API authors can then merge several vendor environments to get the one that they need for their API.

This method will fail if any board is defined in both environments, as it is unclear which one has the correct mapping.

register_backend(*backend: Type[j5.backends.backend.Backend]*) → None
Register a new backend with this environment.

supported_boards
The boards that are supported by this environment.

j5.boards package

Subpackages

j5.boards.arduino package

Submodules

j5.boards.arduino.uno module

Module contents

j5.boards.sb package

Submodules

j5.boards.sb.arduino module

Module contents

j5.boards.sr package

Subpackages

j5.boards.sr.v4 package

Submodules

j5.boards.sr.v4.motor_board module

j5.boards.sr.v4.power_board module

j5.boards.sr.v4.ruggeduino module

j5.boards.sr.v4.servo_board module

Module contents

Module contents

Boards made by Student Robotics.

j5.boards.zoloto package

Submodules

j5.boards.zoloto.camera_board module

Module contents

Submodules

j5.boards.board module

The base classes for boards and group of boards.

class j5.boards.board.**Board**
Bases: object

A collection of hardware that has an implementation.

BOARDS = {}

firmware_version
The firmware version of the board.

static make_all_safe() → None
Make all boards safe.

make_safe() → None
Make all components on this board safe.

name
A human friendly name for this board.

serial_number
The serial number of the board.

static supported_components() → Set[Type[Component]]
The types of component supported by this board.

j5.boards.board_group module

Board Group class.

class j5.boards.board_group.**BoardGroup**(*backend_class: Type[U]*)
Bases: typing.Generic

A group of boards that can be accessed.

backend_class
The Backend that this group uses for Boards.

boards
Get an unordered list of boards in this group.

classmethod get_board_group(*_: Type[T], backend: Type[U]*) → j5.boards.board_group.BoardGroup[~T, ~U][T, U]
Get the board group with the given types.

Whilst the first parameter value is not actually used in the function, we need it for typing purposes. This is similar to how a ProxyType works in Haskell.

make_safe() → None
Make all of the boards safe.

singular() → T
If there is only a single board in the group, return that board.

update_boards() → None
Update the boards in this group to see if new boards have been added.

Module contents

This module contains the boards that we support.

class j5.boards.**Board**
Bases: object

A collection of hardware that has an implementation.

BOARDS = {}

firmware_version
The firmware version of the board.

static make_all_safe() → None
Make all boards safe.

make_safe() → None
Make all components on this board safe.

name
A human friendly name for this board.

serial_number
The serial number of the board.

static supported_components() → Set[Type[Component]]
The types of component supported by this board.

class j5.boards.**BoardGroup**(*backend_class: Type[U]*)
Bases: typing.Generic

A group of boards that can be accessed.

backend_class
The Backend that this group uses for Boards.

boards
Get an unordered list of boards in this group.

classmethod get_board_group(*_: Type[T], backend: Type[U]*) → j5.boards.board_group.BoardGroup[~T, ~U][T, U]
Get the board group with the given types.

Whilst the first parameter value is not actually used in the function, we need it for typing purposes. This is similar to how a ProxyType works in Haskell.

make_safe() → None
Make all of the boards safe.

singular() → T
If there is only a single board in the group, return that board.

update_boards() → None
Update the boards in this group to see if new boards have been added.

j5.components package

Subpackages

j5.components.derived package

Submodules

j5.components.derived.ultrasound module

Module contents

Submodules

j5.components.battery_sensor module

Classes for Battery Sensing Components.

class j5.components.battery_sensor.**BatterySensor**(*identifier: int*, *backend: j5.components.battery_sensor.BatterySensorInterface*)

Bases: *j5.components.component.Component*

A sensor capable of monitoring a battery.

current
Get the current of the battery sensor.

identifier
An integer to identify the component on a board.

static interface_class() → Type[j5.components.battery_sensor.BatterySensorInterface]
Get the interface class that is required to use this component.

voltage
Get the voltage of the battery sensor.

class j5.components.battery_sensor.**BatterySensorInterface**

Bases: *j5.components.component.Interface*

An interface containing the methods required to read data from a BatterySensor.

get_battery_sensor_current(*identifier: int*) → float
Get the current of a battery sensor.

get_battery_sensor_voltage(*identifier: int*) → float
Get the voltage of a battery sensor.

j5.components.button module

Classes for Button.

class j5.components.button.**Button**(*identifier: int*, *backend: j5.components.button.ButtonInterface*)

Bases: *j5.components.component.Component*

A button.

identifier
An integer to identify the component on a board.

static interface_class() → Type[j5.components.button.ButtonInterface]
Get the interface class that is required to use this component.

is_pressed
Get the current pushed state of the button.

wait_until_pressed() → None
Halt the program until this button is pushed.

class j5.components.button.**ButtonInterface**
Bases: *j5.components.component.Interface*

An interface containing the methods required for a button.

get_button_state(*identifier: int*) → bool
Set the state of a button.

wait_until_button_pressed(*identifier: int*) → None
Halt the program until this button is pushed.

j5.components.component module

Base classes for components.

class j5.components.component.**Component**
Bases: object

A component is the smallest logical part of some hardware.

identifier
An integer to identify the component on a board.

static interface_class() → Type[j5.components.component.Interface]
Get the interface class that is required to use this component.

class j5.components.component.**DerivedComponent**
Bases: *j5.components.component.Component*

A derived component is a component that can take another component as a parameter.

For example, a device may be attached to various pins on the board, and this could vary depending on what the user wants. We solve this by passing the pins to the derived component.

```
>>> u = Ultrasound(pin_0, pin_1)
```

identifier
An integer to identify the component on a board.

static interface_class() → Type[j5.components.component.Interface]
Get the interface class that is required to use this component.

class j5.components.component.**Interface**
Bases: object

A base class for interfaces to inherit from.

exception j5.components.component.**NotSupportedByComponentError**
Bases: Exception

This is thrown when hardware does not support the action that is attempted.

j5.components.gpio_pin module

Classes for GPIO Pins.

exception j5.components.gpio_pin.**BadGPIOPinModeError**
Bases: Exception

The pin is not in the correct mode.

class j5.components.gpio_pin.**GPIOPin**(*identifier: int, backend: j5.components.gpio_pin.GPIOPinInterface, *, initial_mode: Union[Type[j5.components.component.DerivedComponent], j5.components.gpio_pin.GPIOPinMode], hardware_modes: Set[j5.components.gpio_pin.GPIOPinMode] = {<GPIOPinMode.DIGITAL_OUTPUT: 3>}, firmware_modes: Set[Type[j5.components.component.DerivedComponent]] = {}*)
Bases: *j5.components.component.Component*

A GPIO Pin.

DEFAULT_FW_MODE = {}

DEFAULT_HW_MODE = {<GPIOPinMode.DIGITAL_OUTPUT: 3>}

analogue_read() → float
Get the scaled analogue reading of the pin.

analogue_write(*new_value: float*) → None
Set the analogue value of the pin.

digital_read() → bool
Get the digital state of the pin.

digital_write(*state: bool*) → None
Set the digital state of the pin.

firmware_modes
Get the supported firmware modes.

identifier
An integer to identify the component on a board.

static interface_class() → Type[j5.components.gpio_pin.GPIOPinInterface]
Get the interface class that is required to use this component.

last_digital_write
Get the last set digital state of the pin.

This does not perform a read operation, it only gets the last set value, which is usually cached in memory.

mode
Get the mode of this pin.

pwm_write(*new_value: float*) → None
Set the PWM value of the pin.

class j5.components.gpio_pin.**GPIOPinInterface**
Bases: *j5.components.component.Interface*

An interface containing the methods required for a GPIO Pin.

get_gpio_pin_digital_state(*identifier: int*) → bool
Get the last written state of the GPIO pin.

get_gpio_pin_mode(*identifier: int*) → j5.components.gpio_pin.GPIOPinMode
Get the hardware mode of a GPIO pin.

read_gpio_pin_analogue_value(*identifier: int*) → float
Read the scaled analogue value of the GPIO pin.

read_gpio_pin_digital_state(*identifier: int*) → bool
Read the digital state of the GPIO pin.

set_gpio_pin_mode(*identifier: int*, *pin_mode: j5.components.gpio_pin.GPIOPinMode*) → None
Set the hardware mode of a GPIO pin.

write_gpio_pin_dac_value(*identifier: int*, *scaled_value: float*) → None
Write a scaled analogue value to the DAC on the GPIO pin.

write_gpio_pin_digital_state(*identifier: int*, *state: bool*) → None
Write to the digital state of a GPIO pin.

write_gpio_pin_pwm_value(*identifier: int*, *duty_cycle: float*) → None
Write a scaled analogue value to the PWM on the GPIO pin.

class j5.components.gpio_pin.**GPIOPinMode**
Bases: enum.IntEnum

Hardware modes that a GPIO pin can be set to.

ANALOGUE_INPUT = 4
The analogue voltage of the pin can be read.

ANALOGUE_OUTPUT = 5
The analogue voltage of the pin can be set using a DAC.

DIGITAL_INPUT = 0
The digital state of the pin can be read

DIGITAL_INPUT_PULLDOWN = 2
Same as DIGITAL_INPUT but internal pull-down is enabled

DIGITAL_INPUT_PULLUP = 1
Same as DIGITAL_INPUT but internal pull-up is enabled

DIGITAL_OUTPUT = 3
The digital state of the pin can be set.

PWM_OUTPUT = 6
A PWM output signal can be created on the pin.

j5.components.led module

Classes for the LED support.

class j5.components.led.**LED**(*identifier: int*, *backend: j5.components.led.LEDInterface*)
Bases: *j5.components.component.Component*

A standard Light Emitting Diode.

identifier
An integer to identify the component on a board.

static interface_class() → Type[j5.components.led.LEDInterface]
Get the interface class that is required to use this component.

state
Get the current state of the LED.

class j5.components.led.**LEDInterface**
Bases: *j5.components.component.Interface*

An interface containing the methods required to control an LED.

get_led_state(*identifier: int*) → bool
Get the state of an LED.

set_led_state(*identifier: int*, *state: bool*) → None
Set the state of an LED.

j5.components.marker_camera module

j5.components.motor module

j5.components.piezo module

j5.components.power_output module

j5.components.servo module

j5.components.string_command module

Module contents

j5.vision package

Submodules

j5.vision.coordinates module

j5.vision.markers module

j5.vision.orientation module

Module contents

Submodules

j5.base_robot module

A base class for robots.

class j5.base_robot.**BaseRobot**
Bases: object

A base robot.

make_safe() → None
Make this robot safe.

exception j5.base_robot.**UnableToObtainLock**
Bases: OSError

Unable to obtain lock.

j5.types module

Useful datatypes that aren't available in the standard lib.

class j5.types.**ImmutableDict**(*members: Mapping[T, U]*)
Bases: typing.Generic

A dictionary whose elements cannot be set.

class j5.types.**ImmutableList**(*members: Union[List[T], Generator[T, None, None]]*)
Bases: typing.Generic

A list whose items cannot be set.

Module contents

j5 Robotics API.

class j5.**BoardGroup**(*backend_class: Type[U]*)
Bases: typing.Generic

A group of boards that can be accessed.

backend_class
The Backend that this group uses for Boards.

boards
Get an unordered list of boards in this group.

classmethod get_board_group(*_: Type[T], backend: Type[U]*) → j5.boards.board_group.BoardGroup[~T, ~U][T, U]
Get the board group with the given types.

Whilst the first parameter value is not actually used in the function, we need it for typing purposes. This is similar to how a ProxyType works in Haskell.

make_safe() → None
Make all of the boards safe.

singular() → T
If there is only a single board in the group, return that board.

update_boards() → None
Update the boards in this group to see if new boards have been added.

class j5.**BaseRobot**
Bases: object

A base robot.

make_safe() → None
Make this robot safe.

class j5.**Environment**(*name: str*)

Bases: object

A collection of backends that can work together.

A number of Backends that we wish to use in a grouping, such as those that all work together in hardware can be added to this group. We can then pass Environments to a Robot object, so that the Robot object can call different methods based on where it is being used.

e.g Hardware Environment

We have *n* boards of *n* different types. They are all physical hardware. We create an Environment containing the Backends to control the physical hardware for our boards.

It is later realised that we want to test code without the physical hardware. We can add Console backends to an environment, and instantiate our Robot object with that environment, so that the console is manipulated rather than the hardware.

This allows for a high degree of code reuse and ensures API compatibility in different situations.

get_backend(*board: Type[Board]*) → Type[j5.backends.backend.Backend]

Get the backend for a board.

get_board_group(*board: Type[BoardT]*) → j5.boards.board_group.BoardGroup[~BoardT, j5.backends.backend.Backend][BoardT, j5.backends.backend.Backend]

Get a board group for the given board type.

merge(*other: j5.backends.environment.Environment*) → None

Merge in the board-backend mappings from another environment.

This allows vendors to predefine Environments and API authors can then merge several vendor environments to get the one that they need for their API.

This method will fail if any board is defined in both environments, as it is unclear which one has the correct mapping.

register_backend(*backend: Type[j5.backends.backend.Backend]*) → None

Register a new backend with this environment.

supported_boards

The boards that are supported by this environment.

Python Module Index

j

Index

F

G

I

J

L

M

N

P

R

S

U

V

W