j5

Release 1.1.2

j5 contributors

May 28, 2023

CONTENTS:

CHAPTER

ONE

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 3.7 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(
```

(continues on next page)

(continued from previous page)

```
            MotorBoard,
            SRV4MotorBoardConsoleBackend,
        )


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␣
```

(continues on next page)

(continued from previous page)

```
↪board.

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


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 Components

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.

1.3.1 Battery Sensor

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

A sensor capable of monitoring a battery.

property current: float

Get the current of the battery sensor.

Returns

current measured by the sensor.

property voltage: float

Get the voltage reported by the battery sensor.

Returns

voltage measured by the sensor.

1.3.2 Button

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

A button.

property is_pressed: bool

Get the current pushed state of the button.

Returns
current pushed state of the button.

wait_until_pressed() → None

Halt the program until this button is pushed.

1.3.3 GPIOPin

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

A GPIO Pin.

analogue_read() → float

Get the scaled analogue reading of the pin.

Returns
scaled analogue reading

analogue_write(*new_value: float*) → None

Set the analogue value of the pin.

Parameters
new_value – analogue value

Raises
ValueError – pin value must be between 0 and 1

digital_read() → bool

Get the digital state of the pin.

Returns
digital read state of the pin.

digital_write(*state: bool*) → None

Set the digital state of the pin.

Parameters
state – digital state.

property firmware_modes: Set[Type[DerivedComponent]]

Get the supported firmware modes.

Returns
supported firmware modes.

property last_digital_write: bool

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.

Returns
last set digital state of the pin

property mode: Union[Type[DerivedComponent], GPIOPinMode]

Get the mode of this pin.

Returns
current mode of the pin.

pwm_write(*new_value: float*) → None

Set the PWM value of the pin.

Parameters
new_value – new duty cycle

Raises
ValueError – pin value must be between 0 and 1

1.3.4 LED

class j5.components.LED(*identifier: int*, *backend: LEDInterface*)

A standard Light Emitting Diode.

property state: bool

Get the current state of the LED.

Returns
current state of the LED.

1.3.5 Motor

class j5.components.Motor(*identifier: int*, *backend: MotorInterface*)

Brushed DC motor output.

property power: Union[float, MotorSpecialState]

Get the current power of this output.

Returns
current power of this output.

1.3.6 Piezo

class j5.components.Piezo(*identifier: int*, *backend: PiezoInterface*, ***, *default_blocking: bool = False*)

A standard piezo.

buzz(*duration: Union[int, float, timedelta]*, *pitch: Union[int, float, Note]*, ***, *blocking: Optional[bool] = None*) → None

Queue a note to be played.

Float and integer durations are measured in seconds.

A buzz can either be blocking, or non-blocking and will fall back to a default if it is not specified.

Parameters

- **duration** – length to play for
- **pitch** – pitch of buzz.
- **blocking** – whether the code waits for the buzz

static verify_duration(*duration: timedelta*) → None

Verify that a duration is valid.

Parameters
duration – duration to validate.

Raises

- **TypeError** – duration must be a timedelta.
- **ValueError** – duration cannot be negative.

static verify_pitch(*pitch: Union[int, float, Note]*) → None

Verify that a pitch is valid.

Parameters
pitch – pitch to validate.

Raises

- **TypeError** – Pitch must be float or Note
- **ValueError** – Frequency must be greater than zero

1.3.7 PowerOutput

class j5.components.**PowerOutput**(*identifier: int*, *backend: PowerOutputInterface*)

A power output channel.

It can be enabled/disabled, and the current being drawn on this channel can be measured.

property current: float

Get the current being drawn on this power output, in amperes.

Returns
current being drawn on this power output, in amperes.

property is_enabled: bool

Get whether the output is enabled.

Returns
output enabled

1.3.8 PWMLED

class j5.components.**PWMLED**(*identifier: int*, *backend: PWMLEDInterface*)

A Light Emitting Diode, driven by a PWM output.

This usually means that the LED is of variable brightness.

property **duty_cycle**: float

Get the current duty cycle of the LED.

Returns
current duty cycle of the LED.

1.3.9 RGBLED

class j5.components.**RGBLED**(*identifier: int*, *backend: RGBLEDInterface*)

A Light Emitting Diode, driven by a PWM output.

This usually means that the LED is of variable brightness.

property **blue**: float

Get the current value of the blue channel.

Returns
current duty cycle of the blue channel.

get_channel(*channel: Union[str, RGBColour]*) → float

Get the current value of a channel.

Parameters
channel – The channel to get the value for.

Returns
The duty cycle for the channel.

Raises
ValueError – channel is not a valid RGB channel.

property **green**: float

Get the current value of the green channel.

Returns
current duty cycle of the green channel.

property **red**: float

Get the current value of the red channel.

Returns
current duty cycle of the red channel.

property **rgb**: Tuple[float, float, float]

Get a tuple of the channel duty cycles.

Returns
tuple of duty cycles (R, G, B).

set_channel(*channel: Union[str, RGBColour]*, *duty_cycle: float*) → None

Set the current value of a channel.

Parameters

- **channel** – The channel to get the value for.
- **duty_cycle** – The duty cycle to set the channel to.

Raises

- **ValueError** – channel is not a valid RGB channel.
- **ValueError** – duty cycle is not in expected range.

1.3.10 Servo

class j5.components.**Servo**(*identifier: int*, *backend: ServoInterface*)

A standard servomotor.

property position: Optional[float]

Get the current position of the Servo.

Returns

current position of the Servo

1.3.11 StringCommand

class j5.components.**StringCommandComponent**(*identifier: int*, *backend: StringCommandComponentInterface*)

A string command component.

This component allows the sending and receiving of commands to a board, so that custom ASCII protocols can be implemented. This is primarily aimed at Boards which can have custom firmware installed by the students that are using them.

execute(*command: str*) → str

Execute the string command and return the result.

This function can be synchronous and blocking.

Parameters

command – command to execute.

Returns

result of command.

Raises

ValueError – command is not valid.

1.3.12 UltrasoundSensor

class j5.components.derived.**UltrasoundSensor**(*gpio_trigger:* GPIOPin, *gpio_echo:* GPIOPin, *backend:* *UltrasoundInterface*, *, *distance_mode:* *bool* = *True*)

Ultrasonic distance sensor.

A sensor that utilises the reflection of ultrasound to calculate the distance to a nearby object.

distance() → Optional[float]

Send a pulse and return the distance to the object.

Returns
Distance measured in metres, or None if it timed out.

Raises
RuntimeError – distance mode is disabled.

pulse() → Optional[timedelta]

Send a pulse and return the time taken.

Returns
Time taken for the pulse, or None if it timed out.

1.4 Supported Hardware

j5 is primarily a framework around which hardware implementations can be built.

However, there are a number of common devices which have implementations provided by j5.

1.4.1 Support Levels

These implementation are split into a number of support levels:

- *Core* - This implementation is part of the core `j5` library.
- *Supported* - This implementation is officially supported by j5.
- *3rd Party* - This implementation is not supported by j5.

1.4.2 Available Integrations

The following integrations are available:

Vendor	Name	Support Level
SourceBots	Arduino Uno Firmware	Core
Student Robotics	KCH v1	Core
Student Robotics	Motor Board v4	Core
Student Robotics	Power Board v4	Core
Student Robotics	Ruggeduino Firmware	Core
Student Robotics	Servo Board v4	Core
Zoloto	Fiducial Marker Pose Estimation	Supported

SourceBots

SourceBots is a not-for-profit organisation aiming to promote Science, Technology, Engineering and Mathematics (STEM) subjects to teenagers. It does this by hosting robotics challenges which encourage participants to work together in an environment markedly different to the way they would at school or college.

SourceBots' kit is largely similar to the Student Robotics kit.

SourceBots Arduino Firmware

Support Level	Core
Bus	USB
Board Class	`j5.boards.sb.SBArduinoBoard`
Console Backend	`j5.backends.console.sb.arduino.SBArduinoConsoleBackend`
Hardware Backend	`j5.backends.hardware.sb.arduino.SBArduinoHardwareBackend`

The SourceBots Arduino firmware is designed for use on an Arduino Uno.

It can be used to control the GPIO pins of the Arduino and measure distances using HC-SR04 ultrasonic sensors.

The following components are available:

- `board.pins` - 18 x GPIOPin
- `board.led` - 1 x LED

Student Robotics

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.

Student Robotics KCH v1

Support Level	Core
Bus	Raspberry Pi GPIO and I2C via Kernel
Board Class	`j5.boards.sr.KCHBoard`

The KCH v1 is a Raspberry Pi HAT designed for the Student Robotics Kit.

The following components are available:

- `board.leds` - A dictionary of RGB LEDs corresponding to the three user controllable LEDs and the start LED.

Student Robotics Power Board v4

Support Level	Core
Bus	USB
Board Class	`j5.boards.sr.v4.PowerBoard`
Console Backend	`j5.backends.console.sr.v4.SRV4PowerBoardConsoleBackend`
Hardware Backend	`j5.backends.hardware.sr.v4.SRV4PowerBoardHardwareBackend`

The Power Board v4 is a board used for managing power in a Student Robotics Kit, and is powered by a LiPo battery.

The following components are available:

- `board.battery_sensor` - A sensor to monitor the LiPo
- `board.outputs` - A dictionary of Power Outputs, indexed by *`j5.boards.sr.v4.PowerOutputPosition`*.
- `board.piezo` - A Piezo buzzer
- `board.start_button` - The start button

The following components also exist, but are not intended for use by competitors:

- `board._error_led` - The red "error" LED
- `board._run_led` - The green "run" LED

Two firmware generations are available for this board, known as the `legacy` (version 3) and `serial` (version 4+) firmwares. Both generations are supported by the backend implementation, which will automatically determine the correct underlying backend to use during the board discovery phase.

Power Board Power Outputs

There are eight total power outputs on the Power Board, 2 high current, 4 low current and 2 5V outputs. The 5V outputs are wired in parallel from the same regulator.

Note: The 5V Regulator is only controllable from board running version 4 "serial" firmware. Additionally, the L2 port is not controllable in version 4 firmware.

---

class j5.boards.sr.v4.**PowerOutputPosition**(*value*)

A mapping of name to number of the PowerBoard outputs.

The numbers here are the same as used in wire communication with the PowerBoard.

FIVE_VOLT = 6

H0 = 0

H1 = 1

L0 = 2

L1 = 3

L2 = 4

L3 = 5

Student Robotics Motor Board v4

Support Level	Core
Bus	USB
Board Class	`j5.boards.sr.v4.MotorBoard`
Console Backend	`j5.backends.console.sr.v4.SRV4MotorBoardConsoleBackend`
Hardware Backend	`j5.backends.hardware.sr.v4.SRV4MotorBoardHardwareBackend`

The Motor Board v4 is a board used for controlling up to two motors.

The following components are available:

- `board.motors` - A list of motors corresponding to the motor outputs.

Student Robotics Servo Board v4

Support Level	Core
Bus	USB
Board Class	`j5.boards.sr.v4.ServoBoard`
Console Backend	`j5.backends.console.sr.v4.SRV4ServoBoardConsoleBackend`
Hardware Backend	`j5.backends.hardware.sr.v4.SRV4ServoBoardHardwareBackend`

The Servo Board v4 is a board used for controlling up to twelve servo motors.

The following components are available:

- `board.servos` - A list of servos corresponding to the servo outputs.

Student Robotics Ruggeduino Firmware

Support Level	Core
Bus	USB
Board Class	`j5.boards.sr.v4.Ruggeduino`
Console Backend	`j5.backends.console.sr.v4.SRV4RuggeduinoConsoleBackend`
Hardware Backend	`j5.backends.hardware.sr.v4.SRV4RuggeduinoHardwareBackend`

The Ruggeduino is a robust microcontroller for IO based on the Arduino Uno.

Student Robotics provides firmware that allows basic control of the Ruggeduino over serial.

The following components are available:

- `board.pins` - 18 x GPIOPin
- `board.led` - 1 x LED

1.5 Extending j5

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.

1.5.1 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.

```
    def set_led_state(self, identifier: int, state: bool) -> None:
        """
        Set the state of an LED.

        :param identifier: identifier of the LED.
        :param state: desired state of the LED.
        """
        raise NotImplementedError  # pragma: no cover


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]:
        """
        Get the interface class that is required to use this component.

        :returns: interface class.
        """
        return LEDInterface

```

1.5.2 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.

        :param identifier: identifier of the LED.
        :returns: current state of the LED.
        """
        raise NotImplementedError  # pragma: no cover
```

1.5.3 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.

```


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

    name: str = "Student Robotics v4 Motor Board"

    def __init__(
        self,
        serial: str,
```

(continues on next page)

(continued from previous page)

```
        backend: Backend,
        *,
        safe_state: MotorState = MotorSpecialState.BRAKE,
    ) -> None:
        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 of the board.

        :returns: Serial number of the board.
        """
        return self._serial

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

        :returns: Firmware version of the board.
        """
        return self._backend.firmware_version
```

1.5.4 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

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

    board = MotorBoard
```

(continues on next page)

(continued from previous page)

```
    @classmethod
    def discover(cls) -> Set[Board]:
        """
        Discover boards that this backend can control.

        :returns: set of 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.

        :returns: serial number reported by the board.
        """
        return self._serial

    @property
    def firmware_version(self) -> Optional[str]:
        """
        The firmware version reported by the board.

        :returns: firmware version reported by the board, if any.
        """
        return None  # Console, so no firmware

    def get_motor_state(self, identifier: int) -> MotorState:
        """
        Get the current motor state.

        :param identifier: identifier of the motor
        :returns: state of the motor.
        """
        # We are unable to read the state from the motor board, in hardware
        # so instead of asking, we'll get the last set value.
```

1.6 Development

This section is about development of *j5*.

1.6.1 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 3.7 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 *ruff* 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`

1.6.2 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.

Slack

We also have casual discussion in `#kit-software` on the Student Robotics Slack.

Meetings

Development of `j5` is discussed in Student Robotics Kit Team meetings.

1.6.3 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 main.
- Go to https://github.com/srobo/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.

- Click publish!
- GitHub Actions will automatically build and upload binaries for the new release to PyPI.

1.6.4 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.

1.6.5 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.

1.6.6 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.

INDEX

V

W