pinout

Release 0.0.20

John Newall

Mar 01, 2022

CONTENTS:

1 Install and Quickstart **3**
- 1.1 Install . . . 3
- 1.2 Quickstart . . . 4

2 Tutorial **7**
- 2.1 Import modules . . . 7
- 2.2 Diagram setup . . . 8
- 2.3 Hardware image . . . 8
- 2.4 Measuring up . . . 8
- 2.5 Add a single pin-label . . . 10
- 2.6 Add Multiple pin-labels . . . 10
- 2.7 Pin-label orientation . . . 11
- 2.8 Title block . . . 12
- 2.9 Legend . . . 12
- 2.10 Export the diagram . . . 13
- 2.11 Next steps . . . 13

3 KiCad integration **15**
- 3.1 Before you start . . . 15
- 3.2 Create a KiCad footprint Library . . . 16
- 3.3 Add an origin . . . 17
- 3.4 Add pin-labels . . . 17
- 3.5 Add an annotation . . . 17
- 3.6 Add a textblock . . . 17
- 3.7 Import KiCad data . . . 18
- 3.8 Template layout . . . 18
- 3.9 Link an image . . . 19
- 3.10 Add labels and Annotations . . . 19
- 3.11 Access text from KiCad . . . 19
- 3.12 Export a diagram . . . 19

4 Modules **21**
- 4.1 Manager . . . 21
- 4.2 Config . . . 22
- 4.3 Core . . . 23
- 4.4 Layout . . . 28
- 4.5 Pin Labels . . . 29
- 4.6 Leaderlines . . . 31
- 4.7 Annotations . . . 32
- 4.8 Legend . . . 33

4.9 Text 34
4.10 Integrated circuits 34

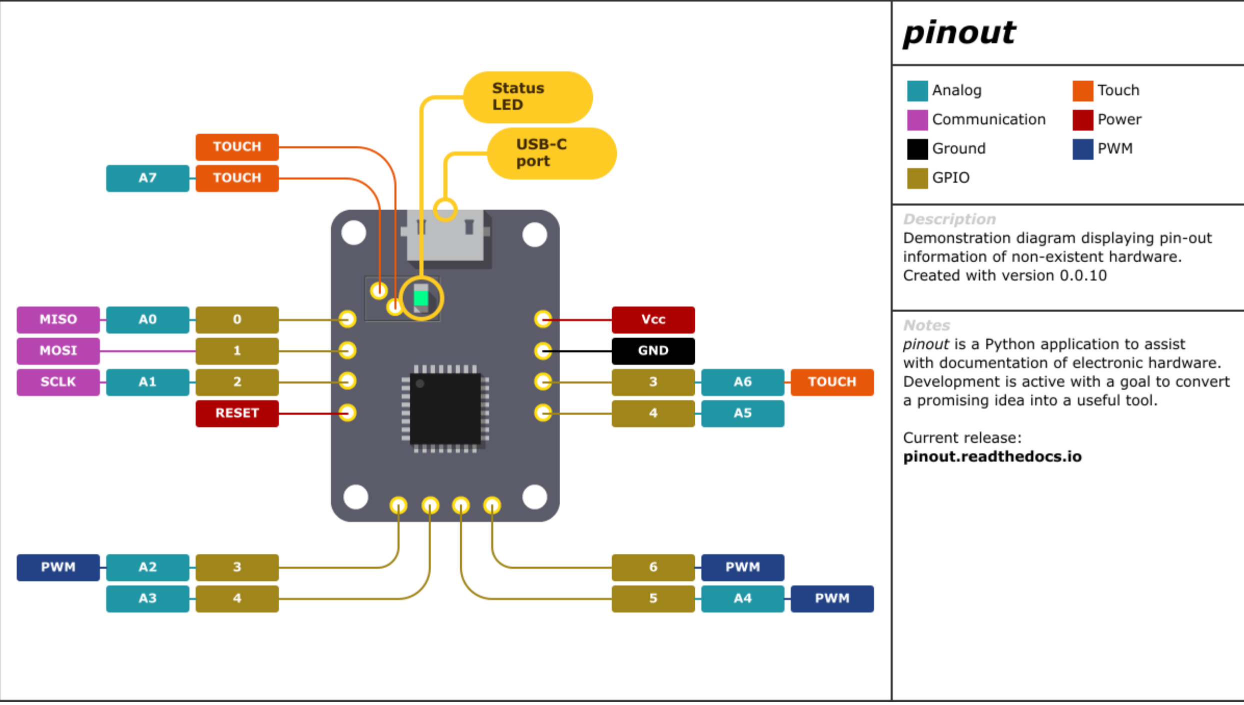
5 Customisation 37
5.1 Stylesheet 37
5.2 Component config 37
5.3 Building components 37

6 Resources 39

7 Indices and tables 41

Index 43

SVG diagram creation from Python code - **pinout** provides an easy method to create pin-out diagrams for electronic hardware.



CHAPTER

ONE

INSTALL AND QUICKSTART

1.1 Install

Using a virtual environment is recommended; Start by installing the *pinout* package from PyPi:

```
pip install pinout

# Or upgrade to the latest version
pip install --upgrade pinout
```

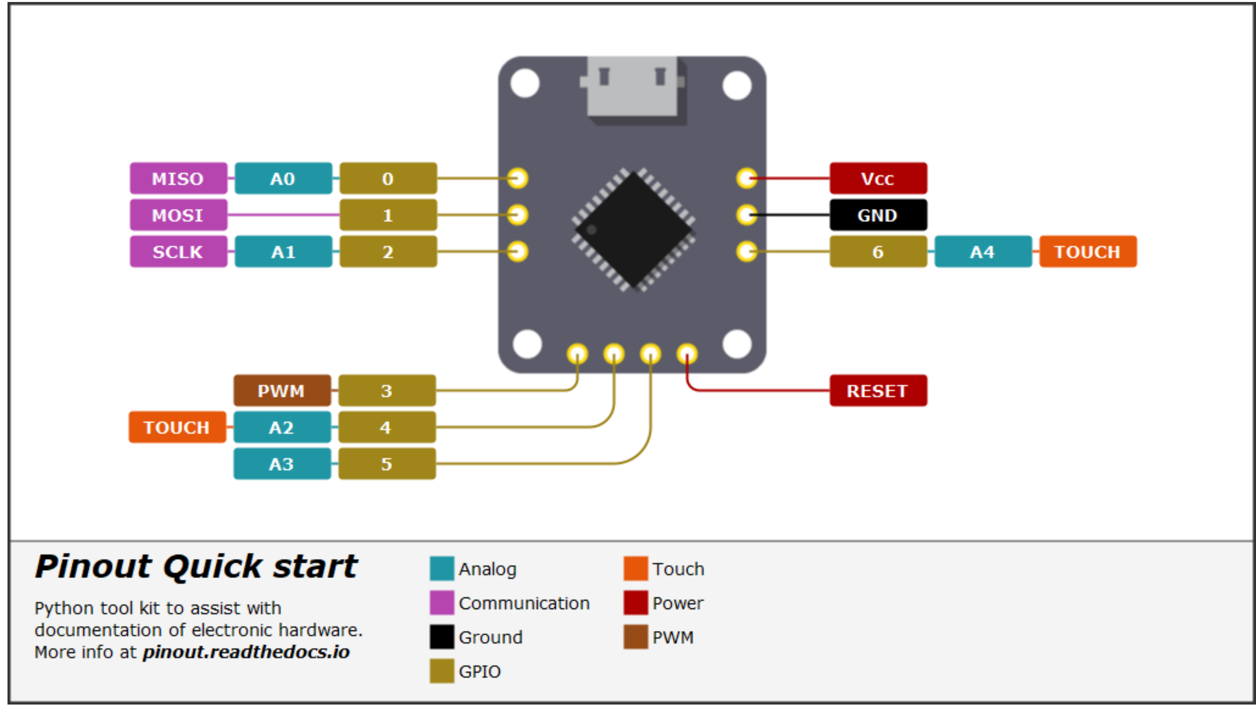
pinout exports diagrams in SVG format and can be used with with no further package installations. With the additional installation of CairoSVG, diagrams can also be exported in PNG, PDF, and PS formats:

```
pip install cairosvg
```

Warning: CairoSVG has non-Python dependencies that will require installing if not present. Installation varies depending on platform and may feel like quite a journey for non-technical users. Information regarding installation requirements can be found in the CairoSVG and Cairo Graphics Library websites.

For Windows users installing GTK3 via MSYS2 may be the most reliable method to install all requirements (Don't forget to add the correct GTK bin folder to the system PATH environmental variable!)

1.2 Quickstart



This guide makes use of a hardware image, stylesheet, data file, and a Python script. Sample files are included with the package and can be duplicated for your use. Open a command line (with enabled virtual environment if you are using one) in the location you plan to work and enter the following

Note: Depending on your operating system the command to invoke Python may differ. This guide uses Windows default method. Exchanging 'py' for 'python' or similar may be required for examples to work on other systems.

```
py -m pinout.manager --duplicate quick_start

# expected output:
# >>> data.py duplicated.
# >>> hardware.png duplicated.
# >>> pinout_diagram.py duplicated.
# >>> styles.css duplicated.
```

Generating the final SVG graphic is done from the command line:

```
py -m pinout.manager --export pinout_diagram.py diagram.svg
```

If everything is correctly configured the newly created file 'diagram.svg' can be viewed in a browser and should look identical to the diagram pictured here.

Warning: Not all SVG viewers are build equal! *pinout* uses SVG format 'under-the-hood' and can also output diagrams in this format. SVG is well supported by modern browsers and applications that *specialize* in rendering SVG such as InkScape. If a *pinout* diagram displays unexpected results (eg. mis-aligned text) cross-check by viewing the diagram in an up-to-date browser (eg. Firefox or Chrome) as an initial trouble-shooting step.

Once you have installed the *pinout* package explore its main features in the *Tutorial*.

CHAPTER

TWO

TUTORIAL

This tutorial walks through the main features available in *pinout*. If you have not installed *pinout* already please read the *Install and Quickstart* section. This tutorial duplicates code from *pinout_diagram.py*. To access a copy of this file and other resources required to complete this tutorial see *Quickstart*.

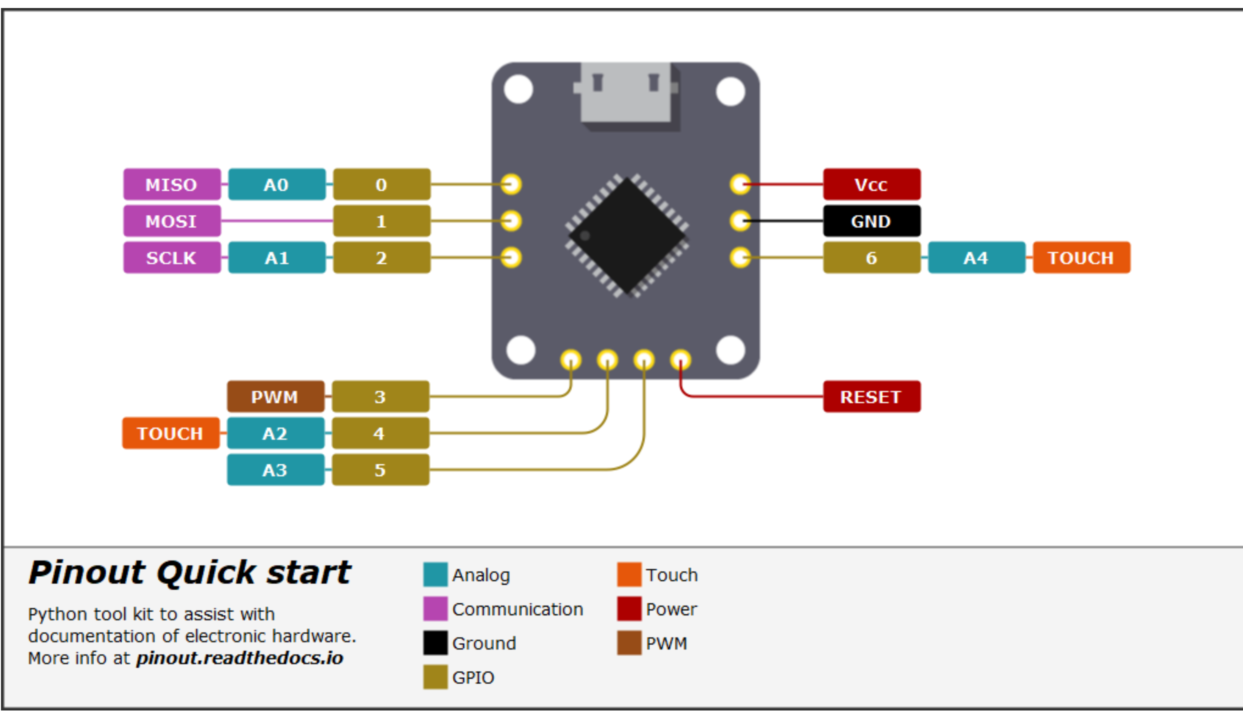


Fig. 1: The finished diagram from this tutorial.

2.1 Import modules

Start by importing pinout modules required to create the sample diagram. For this tutorial the diagram data has been stored in a separate file which is also imported here:

```
from pinout.core import Group, Image
from pinout.components.layout import Diagram_2Rows
from pinout.components.pinlabel import PinLabelGroup, PinLabel
from pinout.components.text import TextBlock
```

(continues on next page)

(continued from previous page)

```
from pinout.components import leaderline as lline
from pinout.components.legend import Legend

# Import data for the diagram
import data
```

2.2 Diagram setup

The *Diagram_2Rows* class creates a blank diagram instance featuring two panels to hold further components and make up the pinout diagram. The instance is named 'diagram' here as this is the default instance name pinout.manager looks for when exporting the final graphic. Presentation styles are controlled via a cascading style-sheet, also added to the diagram here:

```
# Create a new diagram
# The Diagram_2Rows class provides 2 panels,
# 'panel_01' and 'panel_02', to insert components into.
diagram = Diagram_2Rows(1024, 576, 440, "diagram")

# Add a stylesheet
diagram.add_stylesheet("styles.css", embed=True)
```

Components can be grouped independently from a panel. This aids with fine-tuning of a layout as a group of components can be moved as a single unit:

```
# Create a group to hold the actual diagram components.
graphic = diagram.panel_01.add(Group(400, 42))
```

2.3 Hardware image

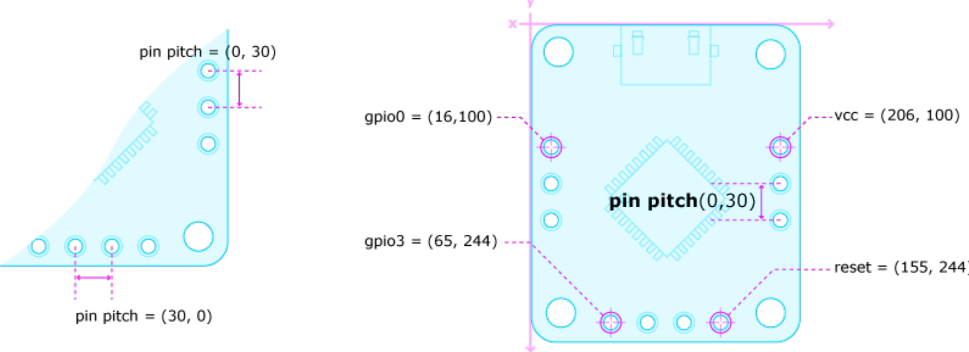
An image that requires pinout information is obviously required and added with the Image class. Width and height arguments are optional. If omitted the pixel dimensions are automatically detected and used. 'x' and 'y' attributes can also be supplied to position the top-left of the image to more suitable coordinates:

```
# Add and embed an image
hardware = graphic.add(Image("hardware.png", embed=True))
```

2.4 Measuring up

Key coordinates on the image need to be documented for related components to align correctly. It is a good idea to undertake measuring as a distinct step and usually quicker than estimating with trial-and-error later on. Measurements are pixel dimensions from the top, left corner *(with an exception - see following note on 'pin pitch')* :

These coordinates could be used 'as is' later on but recording them with the Image class provides a clear association plus coordinates are transformed automatically to remain correctly positioned if the image's x, y, width, or height are adjusted:

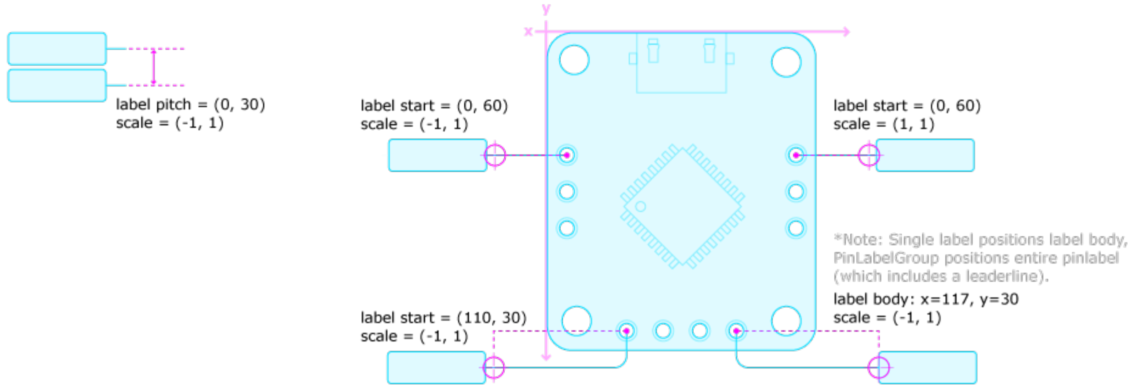



```
# Measure and record key location with the hardware Image instance
hardware.add_coord("gpio0", 16, 100)
hardware.add_coord("gpio3", 65, 244)
hardware.add_coord("reset", 155, 244)
hardware.add_coord("vcc", 206, 100)
# Other (x,y) pairs can also be stored here
hardware.add_coord("pin_pitch_v", 0, 30)
hardware.add_coord("pin_pitch_h", 30, 0)
```

Note: Arbitary (x,y) data can also be recorded with the image. The pin-header pitch has been recorded in this manner. Transformed values can then be automatically calculated if the image's width or height are altered.

pinout provides great flexibility when positioning pin-labels. Key details to note:

- 'x' and 'y' values are **relative** to hardware coordinates.
- label_pitch is an (x,y) offset between each pin-label in a PinLabelGroup
- scale is used to 'flip' labels. Negative values may yield unexpected results
- 'x' and 'y' positions the entire label or row component, *including a leaderline*. A pin-label's body can be positioned in addition to the component positioning.



2.5 Add a single pin-label

In some instances adding pins individually might be appropriate. This pin is being added to the 'graphic' group - the same component that the image is in - and references coordinates filed with the image. Also demonstrated on this pin are some customisations of the pin-label's body and leaderline:

```
# Create a single pin label
graphic.add(
    PinLabel(
        content="RESET",
        x=hardware.coord("reset").x,
        y=hardware.coord("reset").y,
        tag="pwr",
        body={"x": 117, "y": 30},
        leaderline={"direction": "vh"},
    )
)
```

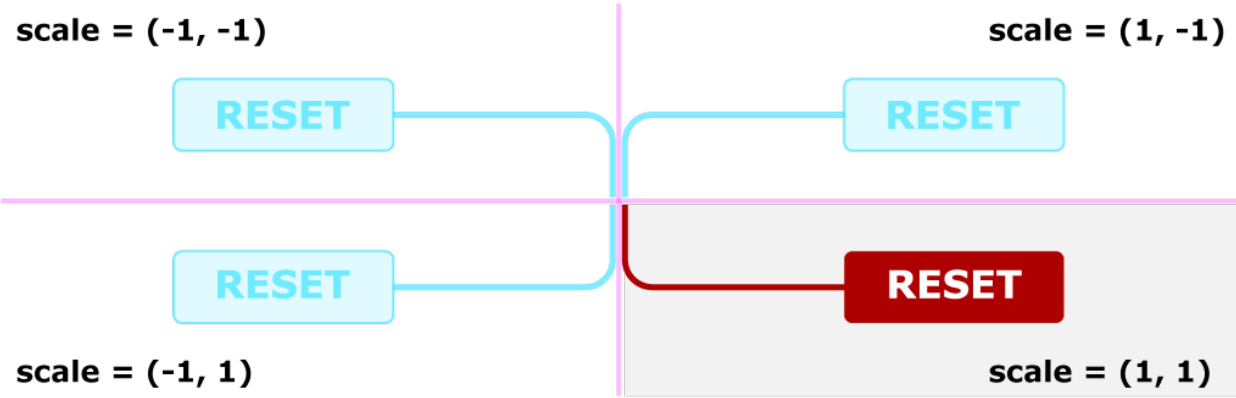
2.6 Add Multiple pin-labels

Where pins are arranged in 'headers' (a line of evenly spaced pins) the PinLabelGroup class can be used to automate many of the geometry calculations required to place individual pin-labels.

- **x, y**: Coordinates of the first pin in the header.
- **pin_pitch**: Distance between each pin of the header. (0, 30) steps 0px right and 30px down for each pin. *TIP*: (30, 0) creates a horizontal header.
- **label_start**: Offset of the first label from the first pin, note that negative x values here may produce unexpected results. pin-label groups should be flipped with scale instead (more explaination later).
- **label_pitch**: Distance between each row of labels.
- **labels**: Label data. See data.py for examples

```
# Create pinlabels on the right header
graphic.add(
    PinLabelGroup(
        x=hardware.coord("vcc").x,
        y=hardware.coord("vcc").y,
        pin_pitch=(0, 30),
        label_start=(60, 0),
        label_pitch=(0, 0),
        labels=data.right_header,
    )
)
```

2.7 Pin-label orientation



SVG format allows 'flipping' or 'mirroring' elements by scaling them with a negative value eg. *scale=(-1, 1)* flips a component around a vertical axis. _pinout_ makes use of this feature, a scale attribute can be supplied to components to flip their layout. This can take some getting use to but provides a concise method of control. The following pin-label groups are scaled to orient in the opposite direction.

```
# Create pinlabels on the left header
graphic.add(
    PinLabelGroup(
        x=hardware.coord("gpio0").x,
        y=hardware.coord("gpio0").y,
        pin_pitch=(0, 30),
        label_start=(60, 0),
        label_pitch=(0, 0),
        scale=(-1, 1),
        labels=data.left_header,
    )
)

# Create pinlabels on the lower header
graphic.add(
    PinLabelGroup(
        x=hardware.coord("gpio3").x,
        y=hardware.coord("gpio3").y,
        scale=(-1, 1),
        pin_pitch=(30, 0),
        label_start=(110, 30),
        label_pitch=(30, 30),
        labels=data.lower_header,
        leaderline=lline.Curved(direction="vh"),
    )
)
```

2.8 Title block

Adding a title and supporting notes can help readers quickly place a diagram in context and summarise important points:

```
# Create a title and a text-block
title_block = diagram.panel_02.add(
    TextBlock(
        data.title,
        x=20,
        y=30,
        line_height=18,
        tag="panel title_block",
    )
)
diagram.panel_02.add(
    TextBlock(
        data.description.split("\n"),
        x=20,
        y=60,
        width=title_block.width,
        height=diagram.panel_02.height - title_block.height,
        line_height=18,
        tag="panel text_block",
    )
)
```

2.9 Legend

Adding a legend is easy as a dedicated component exists in _pinout_. The component flows into multiple columns if a 'max_height' is supplied:

```
# Create a legend
legend = diagram.panel_02.add(
    Legend(
        data.legend,
        x=340,
        y=8,
        max_height=132,
    )
)
```

2.10 Export the diagram

With all the required files present, the diagram can be exported via command-line:

```
py -m pinout.manager --export pinout_diagram.py diagram.svg

# expected output:
# > 'diagram.svg' exported successfully.
```

The exported file is SVG format. When viewed in a web browser it should match the finished diagram shown here. This format is excellent for high quality printing but still an effecient size for web-based usage.

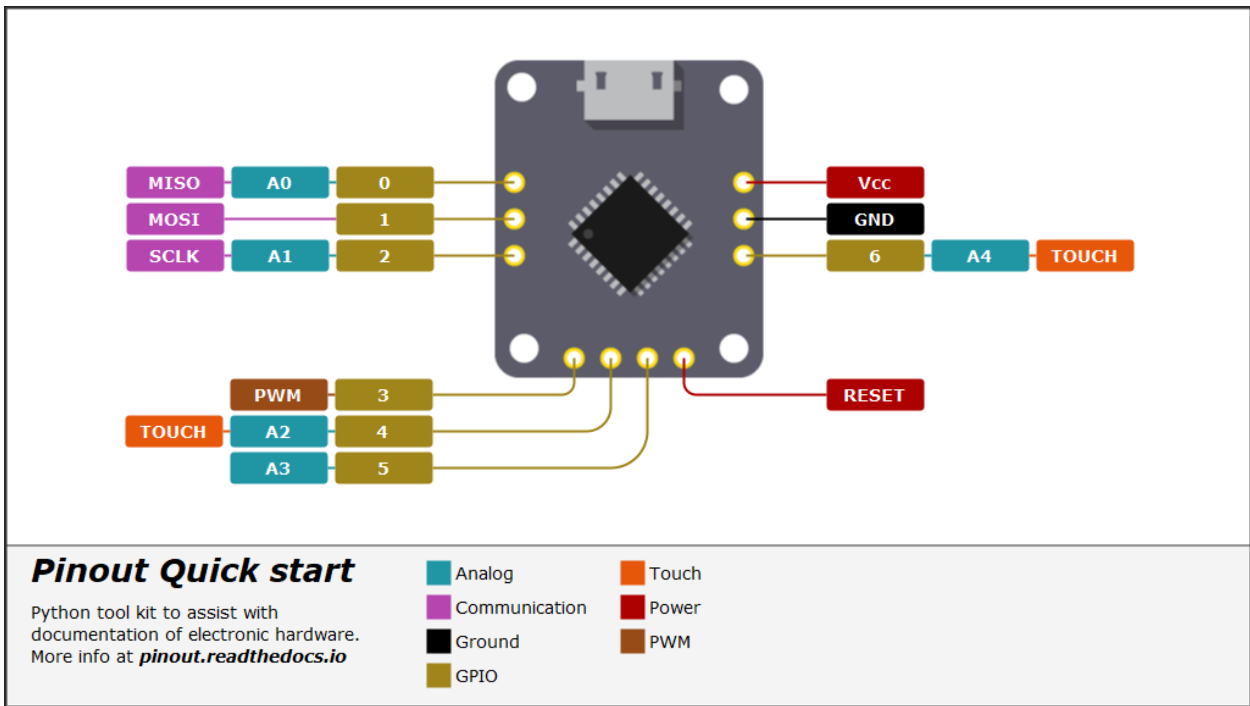


Fig. 2: The finished diagram from this tutorial.

2.11 Next steps

This guide has glossed over many features, attribute, and configurations available. Experimenting with changing values and re-exporting the diagram will quickly reveal their purpose. All function are documented in the *Modules* section.

Depending on you intended usage, linking (instead of embedding) the image might be desirable. Set *embed=False* when adding an image to achieve this outcome. *Note:* When linking, URLs are relative to the exported diagram file. When embedding these URLs are relative to the current working directory (the directory you run the script from).

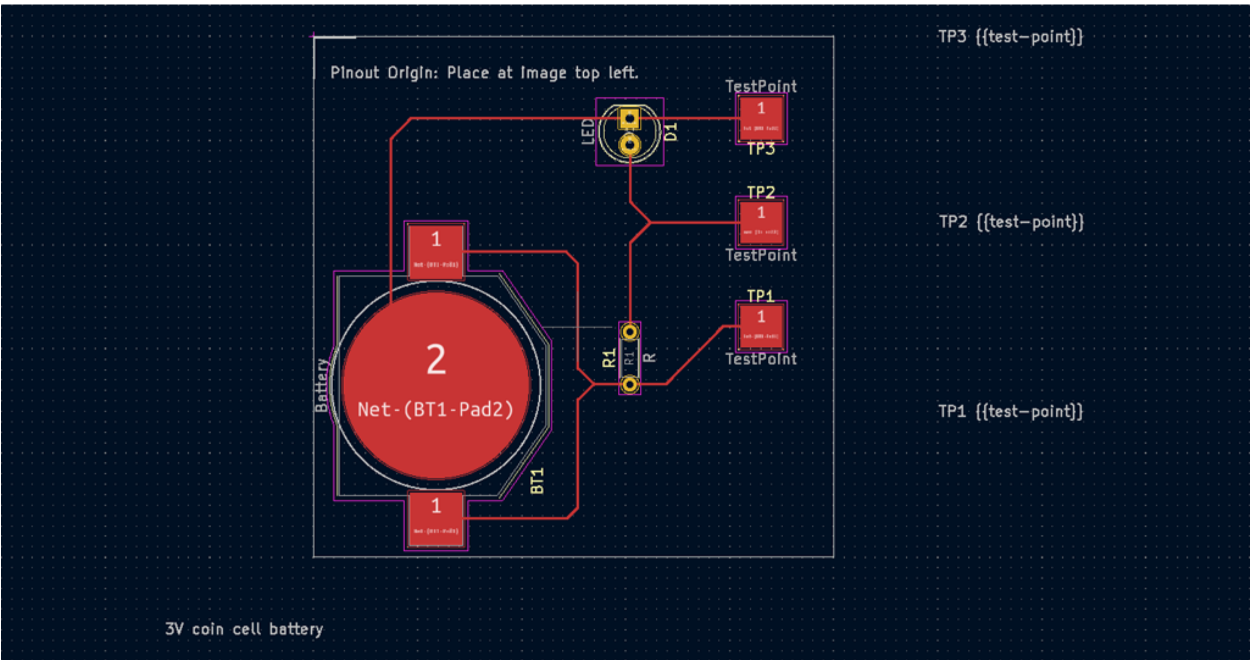
TIP: Embedding the image and stylesheet allows the SVG display correctly in InkScape. This might be an appealing work-flow option for encorporating the diagram into other media or exporting in alternative formats.

More feature-rich examples are available in the samples folder of the pinout github repository.

CHAPTER THREE

KICAD INTEGRATION

pinout provides integration with KiCad (version 5 and 6) allowing users to author pinout content directly onto a PCB design. This allows a better separation of layout and content where KiCad can become a single content source for a diagram template.



3.1 Before you start

Create or obtain (see following note) a KiCad project! This project must include a PCB design which will be enhanced with additional pinout information.

Ensure *pinout* is installed. For more information regarding this step please refer to *Install and quickstart*

Optionally, duplicate the pinout config file. Some KiCad library settings can be customised from this file - Most usefully, the layer that library footprints appear on can be changed:

```
py -m pinout.manager -d config
```

Note: Sample files that demonstrate KiCad (version 6) integration are included with *pinout*. Once duplicated and unzipped, running the python script will export an example diagram:

```
# Duplicate zipped folder with KiCad 6 project and pinout files
py -m pinout.manager -d kicad

# Expected output:
# >>> pinout_kicad_example.zip duplicated.

# Export pinout diagram from unzipped folder
# >>> py -m pinout.manager -e pinout_diagram.py diagram.svg -o

# Expected output:
# >>> 'diagram.svg' exported successfully.
```

3.2 Create a KiCad footprint Library

pinout generates its KiCad footprint library from the command line `py -m pinout.manager <destination folder> <config file> --version <kicad version>`. 'config file' and 'version' are optional. If omitted the version defaults to '6' and default config settings are used:

```
#Example: KiCad 6, saving into the current directory
py -m pinout.manager --kicad_lib .

#Example: KiCad 6, saving into the current directory and referencing a config file
py -m pinout.manager --kicad_lib . config.py

#Example: KiCad 5, saving into the relative directory named 'lib'
py -m pinout.manager --kicad_lib ./lib -v 5

# Expected output:
# >>> pinout footprint library for KiCad created successfully.
```

A folder named *pinout.pretty* will now be present at the location referenced in the command. This folder can be added as a footprint library in your KiCad project.

Note: **KiCad 6**: Footprints are on *User.1* layer by default.

KiCad 5: Footprints are on *Eco1.User* layer by default.

Warning: KiCad 6 allows users to assign an alias to layer names. Only use KiCad's **default** layer names when generating a pinout library.

The pinout footprints can now be added to KiCad like any other footprint library and added to an existing design in the PCB Editor.

3.3 Add an origin

The hardware image used in a diagram must be aligned to KiCad's coordinate system for pinout to successfully align components. This can be done by placing an Origin footprint at a corrosponding location in KiCad. The origin footprint marks where an image's top-left corner will be positioned.

3.4 Add pin-labels

1. Select the PinLabel footprint from the *Choose Footprint* dialogue.
2. Place the footprint at the pin location
3. Move the *Value* text to the desired label location
4. Edit the text value to reflect label content and styling.

Multiple labels can be documented for a single pin by adding additional *text {{tag}}* pairs to the *Value* field. For example this become a row of three labels:

```
GPIO1 {{gpio}} ADC {{analog}} TOUCH {{touch}}
```

By editing the footprint two more fields, that are hidden by default, can be viewed and edited. The *Reference designator* documents the footprints purpose and can be altered without affecting pinout's functions. The additional field is used to document pin-label attributes.

Currently only the pin-label *leaderline* attribute is supported. It can be changed to suit the desired layout and reflects start/end leaderline directions. Valid values are:

- **hh**: horizontal - horizontal
- **vh**: vertical - horizontal

3.5 Add an annotation

Annotations can be added by the same method as pin-labels. 1. Select the Annotation footprint from the *Choose Footprint* dialogue. 2. Place the footprint at the location to be annotated 3. Move the *Value* text to the desired label location 4. Edit the text value to reflect label content and styling.

Tagging the annotation is done with the same 'moustache' style tag *{{tag}}*. The tag text is applied to the final annotation as a css class. Further styling can then be applied via the CSS stylesheet.

By editing the annotation footprint other fields can be accessed and altered - with the same features and limitations - as the PinLabel footprint.

3.6 Add a textblock

A diagram is likely to require text content that is independent from the pinout diagram itself - for instance titles and explainatory notes. To assist with this *pinout* provides the facility to import 'Text items' from KiCad.

KiCad's *Text item* tool is the ideal interface to authoring blocks of text. This tool cannot be used within a footprint but *pinout* collates all Text items that include a moustache-style tag in them. A dictionary is then returned for use within a pinout script. For example:

```
# import kicad pcb data into pinout
kdata = k2p.PinoutParser("kicad6_test.kicad_pcb", dpi=72)

# Retrieve 'Text item' content from KiCad as a dictionary
text = kdata.gr_text()

# Use Text item content to populate a TextBlock
diagram.add(TextBlock(text["txt_tag_01"], tag="txt_tag_01", x=20, y=30))
```

3.7 Import KiCad data

With pinout content documented in KiCad it can now be imported into a *pinout* Python script. The following code snippets are directly from the sample files mentioned at the start of this article. Code for an entire working sample will be duplicated here but descriptions will focus on relevant aspects only.

Both Kicad versions 5 and 6 use the same module. With the module imported a link to the kicad_pcb file can be established:

```
from pinout.core import Group, Image
from pinout.components.layout import Diagram_2Rows
from pinout.components.text import TextBlock
from pinout import kicad2pinout as k2p

# Import KiCad data
kdata = k2p.PinoutParser("kicad_6_pcb/kicad_6_pcb.kicad_pcb", dpi=72, version=6)
```

3.8 Template layout

Whilst labelling can be done in KiCad the overall diagram layout must still be addressed. See the *Tutorial* for more details on this:

```
# Create diagram layout
diagram = Diagram_2Rows(900, 575, 500, tag="diagram")
diagram.add_stylesheet("styles.css")


# Using a 'group' component for easy alignment of all sub-components
graphic = diagram.panel_01.add(Group(300, 65))


# Add an image that corrosponds to the KiCad PCB.
img = graphic.add(Image(src="pcb_graphic.svg", width=300, height=300))
```

3.9 Link an image

Coordinate data from KiCad must be transformed and aligned with the supplied image. This not only translates coordinates to align with the origin footprint but also scales and rotates to remain aligned with an image that has been transformed in *pinout*:

```
# KiCad coordinates will be transformed to match the linked image.
kdata.link_image(img)
```

3.10 Add labels and Annotations

With KiCad data successfully imported and associated with the image it will enhance, adding pin-labels and annotations is easy:

```
 # Add pin-labels and annotations to the 'graphic' group
kdata.add_pinlabels(graphic)
kdata.add_annotations(graphic)
```

3.11 Access text from KiCad

To better separate content and layout *pinout* can also import text content from KiCad. *pinout* scripts can become reuable templates with minimal changes. All text-items that include a 'moustache' style tag are collated into a dict for access in the script. In this example text is used to fill a title block:

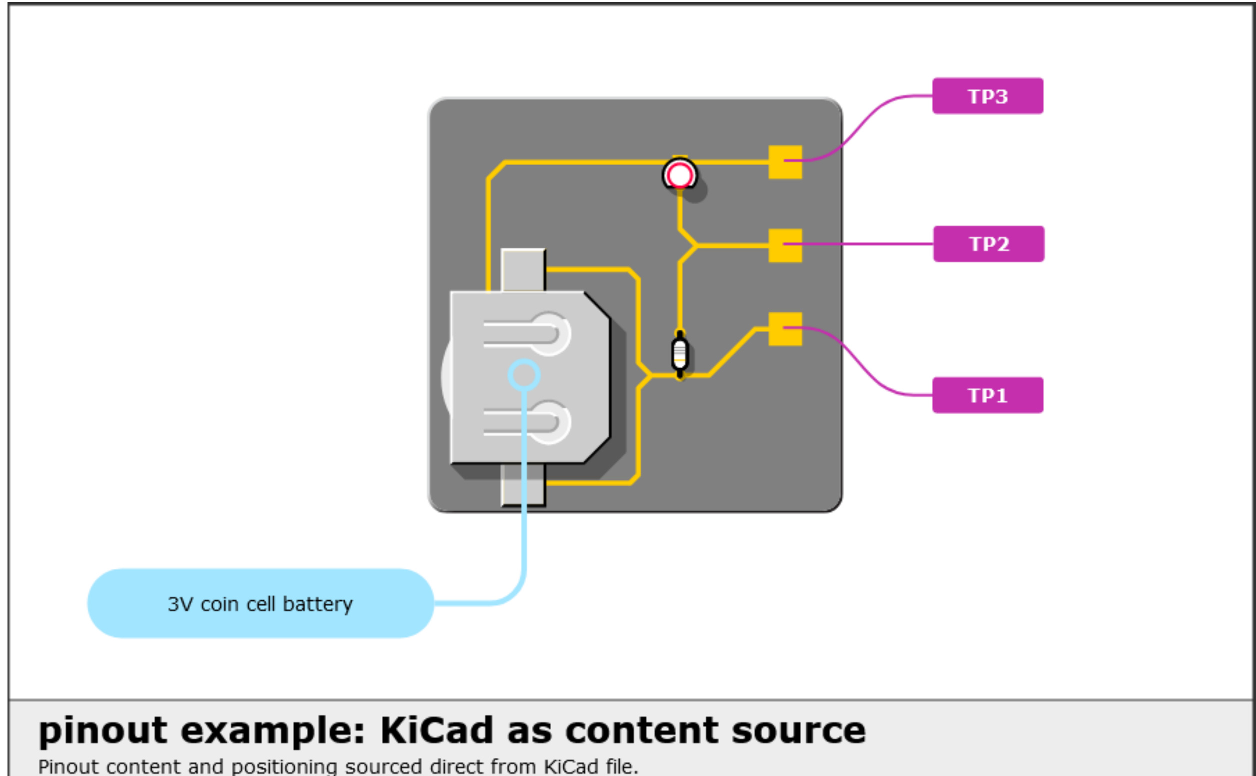
```
# Text from KiCad can be accessed as a dict
textblocks = kdata.gr_text()
diagram.panel_02.add(TextBlock(textblocks["pinout_title"], x=20, y=30))
```

3.12 Export a diagram

The diagram can now be exported in the normal way. For the example script this should go smoothly with predictable results. For other kicad file that include more/different label and tags a revised CSS file needs to be created. *pinout* can provide a reasonable starting point with its auto-styling feature. **Don't forget to update 'add_stylesheet' in the script!**

```
# OPTIONAL EXTRA: Auto generate styles
# >>> py -m pinout.manager --css pinout_diagram.py autostyles.css -o

# Export diagram as SVG:
# >>> py -m pinout.manager -e pinout_diagram.py diagram.svg -o
```


TP3
TP2
TP1
3V coin cell battery
pinout example: KiCad as content source
Pinout content and positioning sourced direct from KiCad file.

CHAPTER

FOUR

MODULES

4.1 Manager

The manager module provides various functions to assist *pinout* create diagrams. For users, Manager is primarily accessed via the command-line for the following.

4.1.1 Duplicate quick_start files

A fast way to get started exploring *pinout* is by trying out the quick_start diagram that is featured in the tutorial. Required files can be duplicated from the *pinout* package via command line:

```
py -m pinout.manager --duplicate quick_start

# expected output:
# >>> data.py duplicated.
# >>> hardware.png duplicated.
# >>> pinout_diagram.py duplicated.
# >>> styles.css duplicated.
```

-d works as a short-hand version of *–duplicate*

4.1.2 Export an SVG diagram

Once a diagram has been documented it can be exported to SVG format via the command-line. Two arguments must be supplied. A path to the diagram python script and destination path including filename:

```
>>> py pinout.manager --export pinout_diagram.py my_diagram.svg

# expected response:
# 'my_diagram.svg' exported successfully.

# Example where pinout.Diagram instance is named 'board_x_diagram'
>>> py pinout.manager --export pinout_diagram.py my_diagram.svg board_x_diagram
```

Details to note:

- *–export* can be expressed as a single letter *-e*
- An *–overwrite* (*-o*) can also be included to overwrite an existing file
- if the instance name is not 'diagram' the alternative name can be added as as third argument

4.1.3 Export in other formats

With the addition of CairoSVG *pinout* is able to export to PNG, PDF, and PS formats. Installation is done via pip:

```
pip install cairosvg
```

Note: CairoSVG has it's own (non-Python) dependencies. See *Install and Quickstart* for more details.

Once these dependencies have been installed replace the filename suffix to export in the desired format:

```
# Export as png
>>> py pinout.manager --export pinout_diagram.py my_diagram.png

# Export as pdf
>>> py pinout.manager --export pinout_diagram.py my_diagram.pdf

# Export as ps
>>> py pinout.manager --export pinout_diagram.py my_diagram.ps
```

4.1.4 Generate a cascading stylesheet

Provided with a diagram file, the manager can extract components and tags, then export a stylesheet based on this data to assist with styling. The resulting stylesheet can then be further edited or a second stylesheet created to supplement the default styles:

```
>>> py pinout.manager --css pinout_diagram.py diagram_styles.css

# expected response:
# Stylesheet created: 'diagram_styles.css'
```

As with exporting an SVG, the *-o* flag can be used to overwrite and existing file. Note, there is no short-hand for the *-css* flag.

4.2 Config

Components with a graphical representation have a variety of configuration attributes that affect their appearance. These attributes can be modified at several location whilst scripting.

4.2.1 Default values

These attributes are stored as Python dictionaries in the **config** module.

A complete set of all default configurations can be duplicated for reference from the command line:

```
py -m pinout.manager --duplicate config

# expected response:
# >>> config.py duplicated.
```

Amending the default configurations can be done by replacing or updating any of the dictionaries with plain Python:

```python
from pinout import config
config.pinlabel["body"].update({"width": 120})

# All pin-label bodies will now default to 120 wide
```

4.2.2 Instance attributes

PinLabels and Annotations accept a dictionary of configurations for some attributes. These values are used to update the default settings for that single instance. This is ideal when small alterations are required for a low number items:

```python
from pinout.core import Diagram
from pinout.components.pinlabel import PinLabel

diagram = Diagram(1200, 675, "pinout")
diagram.add(
    PinLabel(
        x=30,
        y=30,
        tag="sm-label",
        body={"width": 40},
    )
)
```

4.3 Core

4.3.1 Layout

class `pinout.core.`**`Layout`**(*x=0*, *y=0*, *children=None*, ***kwargs*)

Bases: `pinout.core.Component`, `pinout.core.TransformMixin`

Base class fundamentally grouping other components together.

This class is not designed to be used directly. Methods listed here are inherted by child classes and accessible via them.

Parameters

- **x** (*int, optional*) – x-axis location, defaults to 0
- **y** (*int, optional*) – y-axis location, defaults to 0
- **tag** (*string (must be valid css class name), optional*) – css class tag, defaults to None

`render_children`()

Render SVG markup from 'children'

Returns SVG markup

Return type string

4.3.2 StyleSheet

class pinout.core.**StyleSheet**(*src*, *embed=False*)

Bases: `object`

Include a cascading stylesheet.

This class should be added to a diagram via Diagram.add_stylesheet(). Relative paths are relative to the Python script, not the export destination. On export, if the path is relative, it is updated automatically to reflect the destination directory. On export, embedded stylesheets are copied into a <style> tag in the SVG output file.

Parameters

- **path** (*string*) – Path to stylesheet file
- **embed** (*bool, optional*) – Embed stylesheet in exported file, defaults to False

render()

4.3.3 Raw

class pinout.core.**Raw**(*content*)

Bases: `object`

Include arbitary code to the document

Parameters **content** (*string*) – SVG code

4.3.4 Group

class pinout.core.**Group**(*x=0*, *y=0*, *tag=None*, ***kwargs*)

Bases: `pinout.core.Layout`

Group components together

Parameters

- **x** (*int, optional*) – Coordinate of top-left point in x-axis, defaults to 0
- **y** (*int, optional*) – Coordinate of top-left point in y-axis, defaults to 0
- **tag** (*string (must meet css class naming rules), optional*) – CSS class, defaults to None

4.3.5 ClipPath

class pinout.core.**ClipPath**(*children=None*, ***kwargs*)

Bases: `pinout.core.Group`

Define a clip-path component

Once defined the clip-path can be applied to other components by referencing its uuid.

Parameters

- **x** (*int, optional*) – Coordinate of top-left point in x-axis, defaults to 0
- **y** (*int, optional*) – Coordinate of top-left point in y-axis, defaults to 0

- **tag** (*string (must meet css class naming rules), optional*) – CSS class, defaults to None

..automethod:: ClipPath.render

return SVG markup

rtype string

4.3.6 SvgShape

class pinout.core.**SvgShape**(*x=0*, *y=0*, *width=0*, *height=0*, ***kwargs*)

Bases: pinout.core.Component, pinout.core.TransformMixin

Base class for components that have a graphical representation.

Classes that inherit from SvgShape can be considered the smallest building blocks of *pinout*. Along with text components, SvgShape classes represent the actual graphical elements that make up a diagram.

This class has no renderable output itself. Classes that inherit from it must provide their own unique render method. Whist its purpose is primarily as a building block for other classes, SvgShape can be used to reserve an area in components that don't intrinsically have their own width and height (eg Group):

```
from pinout.core import Group, SvgShape

# Create an empty group
grp = diagram.add(Group())
print(f"group dimensions - width:{grp.width}, height:{grp.height}")

# output:
# >>> group dimensions - width:0, height:0

# Add an SvgShape instance to the group
grp.add(SvgShape(x=0, y=0, width=50, height=50))
print(f"group dimensions - width:{grp.width}, height:{grp.height}")

# output:
# >>> group dimensions - width:50.0, height:50.0

# The group now reports a size but does not render anything
```

Parameters

- **x** (*int, optional*) – Location coordinate in x-axis, defaults to 0
- **y** (*int, optional*) – Location coordinate in y-axis, defaults to 0
- **width** (*int, optional*) – Width of the component, defaults to 0
- **height** (*int, optional*) – Height of the component, defaults to 0
- **tag** (*string (must meet css class naming rules), optional*) – CSS class, defaults to None

bounding_coords()

Coordinates representing a shape's bounding-box.

Coordinates are relative to the parent component's origin adn cater for scale and rotation transformations.

Returns (x1, y1, x2, y2)

Return type namedtuple (BoundingCoords)

bounding_rect()

Component's origin coordinates and dimensions

Convenience method that expresses SvgShape.bounding_coords() as coordinates of the shape's origin, width, and height.

Returns (x, y, w, h)

Return type namedtuple (BoundingRect)

4.3.7 Path

class pinout.core.Path(*path_definition='', **kwargs*)

SVG Path object

param path_definition Path definition, defaults to ""

type path_definition str, optional

class pinout.core.Rect(**args, corner_radius=0, **kwargs*)

Bases: *pinout.core.SvgShape*

SVG <rect> object

Parameters **corner_radius** (*int, optional*) – Round rectangle corners, defaults to 0

4.3.8 Text

class pinout.core.Text(*content, **kwargs*)

Bases: *pinout.core.SvgShape*

SVG <text> object

Parameters **content** (*string*) – Text to be included in the tag

4.3.9 Image

class pinout.core.Image(*src, dpi=72, embed=False, **kwargs*)

Bases: *pinout.core.SvgShape*

Include an image in the diagram.

Valid bitmap formats are PNG and JPG - matching the SVG specifications. SVG images can be added via this Image class however they must provided at **1:1 dimensions** and include their own dimensions in the <svg> tag. Additional care needs to be taken when incorporating SVG files as it is possible for CSS classes to clash.

Image size can be controlled by supplying a width and height property. Omiting one, or both, properties results in the supplied image's pixel dimensions to be used.

Where supplied dimensions differ to the image's pixel dimensions the image is scaled proportionally, and centred, to fit supplied dimensions.

Image instances can be added to any component that inherits from the Layout class:

```
from pinout.components.layout import Diagram
from pinout.core import Image

diagram = Diagram(800,400)

# Add an image to the diagram at coordinates (20,20)
diagram.add(Image("hardware.png", x=20, y=20))
```

If an image is to be used multiple times in a single diagram a single instance should be included into the diagram's 'defs' and referenced from there:

```
from pinout.components.layout import Diagram
from pinout.core import Image

diagram = Diagram(800,400)

# Add an image into the diagram's 'defs'
img_src = diagram.add_def(Image("hardware.png"))

# Create x2 new image instances both referencing 'img_src'
img_01 = diagram.add(Image(img_src, x=20, y=20))
img_02 = diagram.add(Image(img_src, x=400, y=20))
```

Parameters

- **path** (*string*) – Path to either an image file on the local file system or a URL. If using a relative path it is relative to the pinout script location.
- **embed** (*bool, optional*) – Embed image in exported file, defaults to False

Coordinates stored in an Image instance can be retrieved with Image.coord(<coord_name>). On retrieval, coordinates are transformed to remain in the correct relative location on image instance regardless of the image's position, width, height, and rotation, for example:

```
from pinout.components.layout import Diagram
from pinout.core import Image

diagram = Diagram(800,400)

# Create an Image instance 'img'
# Parameters match desired output and may
# differ from the image's actual dimensions
img = diagram.add(Image(
    "hardware.png",
    x=50,
    y=10,
    width=100,
    height=200,
    rotate=30
))

# Add a coordinate to 'img'
# This coordinate is measured against the original image at 1:1 scale
```

(continues on next page)

(continued from previous page)

```
img.add_coord("pin_a", 110, 150)

# The transformed coordinate aligns correctly on the transformed image
pin_a = img.coord("pin_a")
```

By default returned coordinates include any offset that occurs when non-proportional width and height are set. By setting *raw=True* the coordinates are scaled purely on actual size vs. user nominated size. This is useful for documenting *pin_pitch*.

Parameters

- **name** (*string*) – Name of coordinate
- **raw** (*bool, optional*) – Return a scaled values without image offset, defaults to False

Returns Coordinates scaled to match image scaling

Return type tuple (x, y)

4.4 Layout

4.4.1 Diagram

class pinout.components.layout.**Diagram**(*width*, *height*, *tag=None*, ***kwargs*)

Bases: `pinout.core.Layout`

Basis of a pinout diagram

Parameters

- **width** (*int*) – width of diagram
- **height** (*int*) – height of diagram
- **tag** (*string (must comply to CSS naming rules), optional*) – CSS class applied to diagram, defaults to None

add_stylesheet(*path*, *embed=False*)

Add a stylesheet to the diagram

Pinout relies on cascading-style-sheet (CSS) rules to control presentation attributes of components.

The path attribute is dependent on whether the styles are linked or embedded. When linked, the path is relative to the exported file. When embedded the path is relative to the diagram script file.

Parameters

- **path** (*string*) – Path to stylesheet file
- **embed** (*bool, optional*) – embed stylesheet in exported file, defaults to True

render()

Render children into an <svg> tag.

Returns SVG markup

Return type string

4.4.2 Panel

class pinout.components.layout.**Panel**(*width*, *height*, *inset=None*, ***kwargs*)

Bases: *pinout.core.Layout*

The basic building block to control layout (grouping and location) of components that make up a complete diagram document. The Panel component renders two rectangles - and outer and inner rectangle - behind all child components to assist with graphical styling.

The inset attribute controls dimensions of the 'inner rectangle'. All children are aligned relative to the inset coordinate (x1, y1).

The inner dimensions can be accessed via the properties `Panel.inset_width` and `Panel.inset_height`.

Parameters

- **width** (*int*) – Width of component
- **height** (*int*) – Height of component
- **inset** (*Tuple (x1, y1, x2, y2), optional*) – Inset of inner dimensions, defaults to None

4.5 Pin Labels

4.5.1 Base

class pinout.components.pinlabel.**Base**(*content=''*, *x=0*, *y=0*, *tag=None*, *body=None*, *leaderline=None*, ***kwargs*)

Bases: *pinout.core.Group*

Label component designed specifically for labelling pins.

Parameters

- **content** (*str, optional*) – Text displayed in label, defaults to ""
- **x** (*int, optional*) – position of label on x-axis , defaults to 0
- **y** (*int, optional*) – position of label on y-axis, defaults to 0
- **tag** (*str (CSS name compliant), optional*) – categorise the label - applied as a CSS class, defaults to None
- **body** (*dict or pinlabel.Body instance, optional*) – replace or configure the default body component, defaults to None
- **leaderline** (*dict or pinlabel.Leaderline, optional*) – replace or configure the default leaderline component, defaults to None

4.5.2 PinLabel

class pinout.components.pinlabel.**PinLabel**(*content=''*, *x=0*, *y=0*, *tag=None*, *body=None*, *leaderline=None*, ***kwargs*)

Bases: *pinout.components.pinlabel.Base*

See Base for details of this component.

4.5.3 Body

class pinout.components.pinlabel.**Body**(*x*, *y*, *width*, *height*, *corner_radius=0*, ***kwargs*)

Bases: *pinout.core.SvgShape*

Graphical shape that makes up the body of a pinlabel.

Parameters

- **x** (*int*) – position of label on x-axis
- **y** (*int*) – position of label on y-axis
- **width** (*int*) – Width of label body
- **height** (*int*) – Height of label body
- **corner_radius** (*int, optional*) – Corner radius of label body, defaults to 0

4.5.4 Leaderline

class pinout.components.pinlabel.**Leaderline**(*direction='hh'*, ***kwargs*)

Bases: *pinout.components.leaderline.Curved*

Graphical line joining the label origin coordinates to the label body.

Parameters **lline** (*dict of leaderline attributes or replacement Leaderline instance*) – Override configuration or replace the pinlabel's leaderline.

4.5.5 PinLabelGroup

class pinout.components.pinlabel.**PinLabelGroup**(*x*, *y*, *pin_pitch*, *label_start*, *label_pitch*, *labels*, *leaderline=None*, *body=None*, ***kwargs*)

Bases: *pinout.core.Group*

Convenience class to place multiple rows of pin-labels on a pin-header.

This is the recommended method of adding pin labels to a diagram. Locate the PinLabelSet by setting *x* and *y* attributes.

Pitch is the distance, in pixels, between each pin of the header. (0, 30) steps 0px right and 30px down for each pin. (30, 0) creates a horizontal header. (-30, 0) creates a horizontal header in the reverse direction. This can be useful for 'stacking' rows in reversed order to avoid leader-lines overlapping.

Parameters

- **x** (*int*) – x-coordinate of the first pin in the header
- **y** (*int*) – y-coordinate of the first pin in the header
- **pin_pitch** (*tuple: (x,y)*) – Distance between pins in the header

- **label_start** (*tuple: (x,y)*) – Offset of the first label from the first pin
- **label_pitch** (*tuple: (x,y)*) – Distance between each row of labels
- **labels** (*List*) – Label data
- **leaderline** (*dict or Leaderline object, optional*) – Leaderline customisations, defaults to None
- **body** (*dict or LabelBody object, optional*) – Label body customisations, defaults to None

4.6 Leaderlines

4.6.1 Leaderline

class pinout.components.leaderline.**Leaderline**(*direction='hh'*, ***kwargs*)

Bases: *pinout.core.Path*

Leaderline base object.

Parameters **direction** (*str, optional*) – 2 letter code, defaults to "hh"

The leaderline connects an origin and destination point. Route taken is controlled with a *direction* argument where the first character dictates the start direction and the second character the end direction:

- **vh**: vertical , horizontal
- **hv**: horizontal , vertical
- **hh**: horizontal , horizontal
- **vv**: vertical , vertical

end_points(*origin*, *destination*)

Locate origin and destination coordinates.

The end_point method takes two components as arguments and returns coordinates that are aligned with the centre coordinates of the relevant side.

Parameters

- **origin** (*component with width and height attributes and bounding_coords method*) – origin component
- **destination** (*component with width and height attributes and bounding_coords method*) – destination component

Returns coordinates of start and end points

Return type Tuple ((ox, oy), (dx, dy))

4.6.2 Curved

class pinout.components.leaderline.**Curved**(*direction='hh'*, ***kwargs*)

Bases: `pinout.components.leaderline.Leaderline`

Leaderline comprised of one or two curved corners.

4.6.3 Angled

class pinout.components.leaderline.**Angled**(*direction='hh'*, ***kwargs*)

Bases: `pinout.components.leaderline.Leaderline`

Leaderline comprised of one or two sharp 90 degree corners.

4.6.4 Straight

class pinout.components.leaderline.**Straight**(*direction='hh'*, ***kwargs*)

Bases: `pinout.components.leaderline.Leaderline`

Leaderline comprised of a single straight line.

4.7 Annotations

4.7.1 Annotation

class pinout.components.annotation.**AnnotationLabel**(*content=None*, *body=None*, *leaderline=None*, *target=None*, ***kwargs*)

Bases: `pinout.core.Group`

Annotation style label.

An alternative method to 'label' a diagram, suitable for highlighting hardware details.

It is likely the body, leaderline, and target will all require customisation to best suit specific usages. Several methods of customisation are possible:

diagram-wide customisations:

- Over-ride dictionary settings in pinout.config.annotation
- Over-ride default annotation body, leaderline, and target classes

instance specific customisations:

- Supply a dictionary of arguments to body, content, leaderline, and target attributes. These override config.annotation settings
- provide an alternative component instance to body, content, leaderline, and target attributes.

Parameters

- **content** (*[type]*) – [description]
- **body** (*[type], optional*) – [description], defaults to None
- **leaderline** (*[type], optional*) – [description], defaults to None
- **target** (*[type], optional*) – [description], defaults to None

4.7.2 Body

class pinout.components.annotation.**Body**(**args*, *corner_radius=0*, ***kwargs*)

Bases: *pinout.core.Rect*

4.7.3 Content

class pinout.components.annotation.**Content**(*content*, *line_height=None*, ***kwargs*)

Bases: *pinout.components.text.TextBlock*

Content can be provided as a string, list, dictionary, or component instance. Strings are presented as a single line. Entries of a list present as lines of text. If a dictionary is provided it updates the default config settings and expects the 'content' attribute to be a list.

4.7.4 Leaderline

class pinout.components.annotation.**Leaderline**(*direction='hh'*, ***kwargs*)

Bases: *pinout.components.leaderline.Curved*

4.7.5 Target

class pinout.components.annotation.**Target**(**args*, *corner_radius=0*, ***kwargs*)

Bases: *pinout.core.Rect*

4.8 Legend

4.8.1 Legend

class pinout.components.legend.**Legend**(*data*, *max_height=None*, ***kwargs*)

Auto generate a legend component

Note: *pinout* does not calculate text widths. A manually provided width may be required to ensure text remains enclosed within the legend.

Parameters

- **data** (*[type]*) – [description]
- **max_height** (*[type], optional*) – [description], defaults to None

4.8.2 Swatch

class pinout.components.legend.**Swatch**(*width=None*, *height=None*, ***kwargs*)

Graphical icon for display in LegendEntry

Parameters

- **width** (*int, optional*) – Width of swatch, defaults to None
- **height** (*int, optional*) – Height of swatch, defaults to None

4.8.3 Entry

`class pinout.components.legend.LegendEntry(content, width=None, height=None, swatch=None, **kwargs)`

Legend entry comprised of a swatch and single line of text.

The swatch attribute accepts either a dictionary of Swatch attributes or a Swatch instance. Swatch styling (ie filling with color) is done via CSS and should reference the LegendEntry class(es).

Parameters

- **content** (*[type]*) – Text displayed in entry
- **width** (*int, optional*) – Width of entry, defaults to None
- **height** (*int, optional*) – height of entry, defaults to None
- **swatch** (*dict or Swatch, optional*) – Graphical icon included in entry, defaults to None

4.9 Text

4.9.1 TextBlock

`class pinout.components.text.TextBlock(content, line_height=None, **kwargs)`

Bases: `pinout.core.Group`

Multiline text component.

The TextBlock accepts either a string or list for content. Each list entry is presented as a line of text. Where a string is provided, it is converted to a list by splitting on new-line characters ('\n') and stripping whitespace from start and end of each line created.

Note: *pinout* cannot detect text character size! Consequently care should be taken to ensure text does not render outside expected boundaries.

Parameters

- **content** (*String or List*) – Text to be displayed
- **line_height** (*int, optional*) – Distance between lines, defaults to None

4.10 Integrated circuits

pinout can generate simple integrated circuit (IC) graphics - Ideal for documenting stand-alone IC components.

DIP and QFP components can be utilised in a diagram in the same way as an image. However helper functions also exists for easy application of labels to these component.

4.10.1 Labelled QFP graphic

pinout.components.integrated_circuits.**labelled_qfn**(*labels*, *length=160*, *label_start=(100, 20)*, *label_pitch=(0, 30)*)

Generate a QFP graphic with pin-labels applied.

Parameters

- **labels** (*list*) – List of label data
- **length** (*int, optional*) – length of the IC sides (including legs), defaults to 160
- **label_start** (*tuple, optional*) – Offset of the first label from the first pin, defaults to (100, 20)
- **label_pitch** (*tuple, optional*) – Offest between each label row, defaults to (0, 30)

Returns IC graphic with pinlabels applied

Return type SVG markup

4.10.2 Labelled DIP graphic

pinout.components.integrated_circuits.**labelled_dip**(*labels*, *width=100*, *height=160*, *label_start_x=100*, *label_pitch=(0, 30)*)

Generate a DIP graphic with pin-labels applied.

Parameters

- **labels** (*list*) – List of label data
- **width** (*int, optional*) – Width of IC (includes legs), defaults to 100
- **height** (*int, optional*) – Height of IC (includes inset), defaults to 160
- **label_start_x** (*int, optional*) – Offset in x-axis of first label from first pin, defaults to 100
- **label_pitch** (*tuple, optional*) – Offest between each label row, defaults to (0, 30)

Returns IC graphic with pinlabels applied

Return type SVG markup

4.10.3 Dual in-line package (DIP)

class pinout.components.integrated_circuits.**DIP**(*pin_count*, *width*, *height*, ***kwargs*)

Bases: *pinout.core.Group*

Create a dual in-line package graphic

Parameters

- **pin_count** (*int*) – Total number of pins on the integrated circuit
- **width** (*int*) – width of the graphic, including body and legs
- **height** (*int*) – height of the graphic, including body and legs

Dimensions can be modified to depict a variety of IC types, eg SOIC and TSOP.

Parameters

- **index** (*int*) – Pin number (starts at 1)
- **rotate** (*bool, optional*) – If true, includes component rotation in the calculation, defaults to True

Returns coordinates of the pin relative to the IC's origin

Return type namedtuple (x,y)

4.10.4 Quad flat package (QFP)

class pinout.components.integrated_circuits.**QFP**(*pin_count*, *length*, ***kwargs*)

Bases: *pinout.core.Group*

Create a quad flat package graphic

Parameters

- **pin_count** (*int*) – Total number of pins on the integrated circuit
- **length** (*int*) – length of the QFP sides

Dimensions can be modified to depict a variety of 'quad' IC types.

Parameters

- **index** (*int*) – Pin number (starts at 1)
- **rotate** (*bool, optional*) – If true, includes component rotation in the calculation, defaults to True

Returns coordinates of the pin relative to the IC's origin

Return type namedtuple (x,y)

CHAPTER

FIVE

CUSTOMISATION

Documentation conveys not just information about its subject but also the personality of the owner. In the context of product/electronics documentation this 'personality' may be of the hardware itself, the creator of the hardware, or company that creates/distributes/sells the hardware.

Many *pinout* components have facility for customisation and easy integration into a diagram.

5.1 Stylesheet

The first stop for altering a diagram's appearance is to edit its stylesheet. Presentation styles are all controlled here. If you are coming with some knowlege of CSS for web, be aware SVG has some different names for rules!

5.2 Component config

Altering the geometry of default components can be done by changing, or providing new, config values. See the *Config* section for more details

5.3 Building components

It is possible to build new components and integrate them into *pinout*.

Existing components are split into parts to allow easier overriding. Where a universal change is desired this maybe the best approach - until a guide is written for this, reviewing the package code (hosted on github) is recommended.

Insertion of customised elements into some component instances is also possible and suitable where only small changes, or multiple variants, of a component are required in a single diagram.

PinLabel has 'leaderline' and 'body' attributes. These accept either a dictionary of values (see *Config*) or an instance that will be used in preference to the equivalent default component.

Annotation has 'leaderline', 'body', and 'target' atttributes that accept new component instances.

An example: The following code can be added to the quick_start script ('pinout_diagram.py') for quick and easy testing:

```
# Import required modules and class at top of the script
from pinout.components import pinlabel
from pinout.core import Path

```

(continues on next page)

(continued from previous page)

```
# Create a new pin-label body class
# and override the render function
class SkewLabelBody(pinlabel.Body):
    def render(self):
        skew = 3
        path_def = " ".join(
            [
                f"M {self.x + skew} {self.y -self.height/2}",
                f"l {self.width} 0",
                f"l {-skew*2} {self.height}",
                f"l {-self.width} 0" "Z",
            ]
        )
        body = Path(path_definition=path_def, tag="label__body")
        return body.render()


# Insert the following before the export statement
# Add an instance of the custom pin-label body to the diagram
diagram.add(
    pinlabel.PinLabel(
        content="SKEWED",
        x=50,
        y=50,
        body=SkewLabelBody(70, 0, 100, 30),
    )
)
```

CHAPTER

SIX

RESOURCES

Every *pinout* diagram has the fundamental requirement of an image to enhance with graphical additions. This prerequisite can be sizable barrier to creating the clearest diagram possible.

During *pinout* development several image create methods have been investigated. Community members have also reached out and generously provided feedback and further options to tryout.

Documented here is a list of what was tried during *pinout* development and some community input (some tried, some on my list to try out). The intent is to revisit all option and write-up some reviews and process notes.

- Export from KiCad
- Create from scratch + InkScape + Illustrator + With exported elements from KiCad
- Photograph
- https://github.com/yaqwsx/PcbDraw
- fritzing

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

A

add_stylesheet() (*pinout.components.layout.Diagram method*), 28
Angled (*class in pinout.components.leaderline*), 32
AnnotationLabel (*class in pinout.components.annotation*), 32

B

Base (*class in pinout.components.pinlabel*), 29
Body (*class in pinout.components.annotation*), 33
Body (*class in pinout.components.pinlabel*), 30
bounding_coords() (*pinout.core.SvgShape method*), 25
bounding_rect() (*pinout.core.SvgShape method*), 26

C

ClipPath (*class in pinout.core*), 24
Content (*class in pinout.components.annotation*), 33
Curved (*class in pinout.components.leaderline*), 32

D

Diagram (*class in pinout.components.layout*), 28
DIP (*class in pinout.components.integrated_circuits*), 35

E

end_points() (*pinout.components.leaderline.Leaderline method*), 31

G

Group (*class in pinout.core*), 24

I

Image (*class in pinout.core*), 26

L

labelled_dip() (*in module pinout.components.integrated_circuits*), 35
labelled_qfn() (*in module pinout.components.integrated_circuits*), 35
Layout (*class in pinout.core*), 23
Leaderline (*class in pinout.components.annotation*), 33
Leaderline (*class in pinout.components.leaderline*), 31
Leaderline (*class in pinout.components.pinlabel*), 30
Legend (*class in pinout.components.legend*), 33
LegendEntry (*class in pinout.components.legend*), 34

P

Panel (*class in pinout.components.layout*), 29
Path (*class in pinout.core*), 26
PinLabel (*class in pinout.components.pinlabel*), 30
PinLabelGroup (*class in pinout.components.pinlabel*), 30

Q

QFP (*class in pinout.components.integrated_circuits*), 36

R

Raw (*class in pinout.core*), 24
Rect (*class in pinout.core*), 26
render() (*pinout.components.layout.Diagram method*), 28
render() (*pinout.core.StyleSheet method*), 24
render_children() (*pinout.core.Layout method*), 23

S

Straight (*class in pinout.components.leaderline*), 32
StyleSheet (*class in pinout.core*), 24
SvgShape (*class in pinout.core*), 25
Swatch (*class in pinout.components.legend*), 33

T

Target (*class in pinout.components.annotation*), 33
Text (*class in pinout.core*), 26
TextBlock (*class in pinout.components.text*), 34