Programmering af Mobile Robotter

RB1-PMR – Module 4: Object-Oriented Programming (OOP) and Actuator Interface

Jes Hundevadt Jepsen | jegj@mmmi.sdu.dk
SDU Drone Center, MMMI, University of Southern Denmark, Odense, Denmark

Agenda

- Recap of last module
- Object-Oriented Programming (OOP) concepts
- Actuators
  - DC motors
    - H-Bridge
    - Pulse-Width Modulation (PWM)
  - Servo motors
  - Stepper motors
- Introduction to Extra Credit Activities #2: Stepper Motor Controller class
- Assignments

Recap

- Recap of last module
- **Hardware walk-through**
  - RP2040, Raspberry Pi Pico, Monk Makes, etc.
- **Software walk-through**
  - Flash firmware, Thonny Python IDE, etc.
- **Introduction to IO programming** using MicroPython
  - Standard **libraries** and **micro-libraries**
  - Key modules, classes and functions
    - Work with the Machine, Pin, and Timer modules
- **Assignments** and **Portfolio 1**
- **Soldered the Practice PCB**
  - Which we will be using for some of the assignments as well

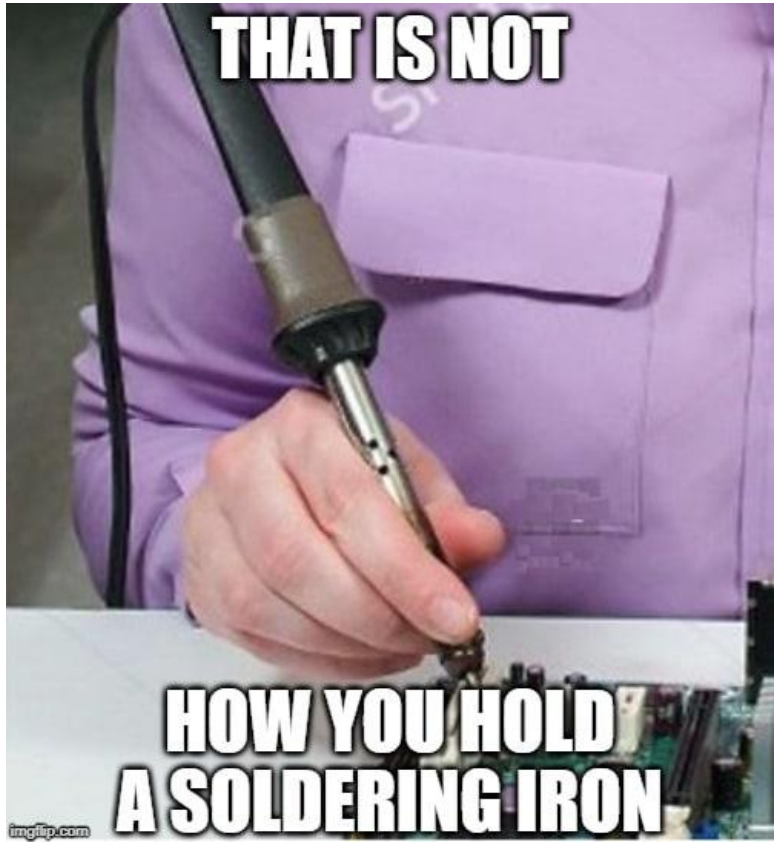
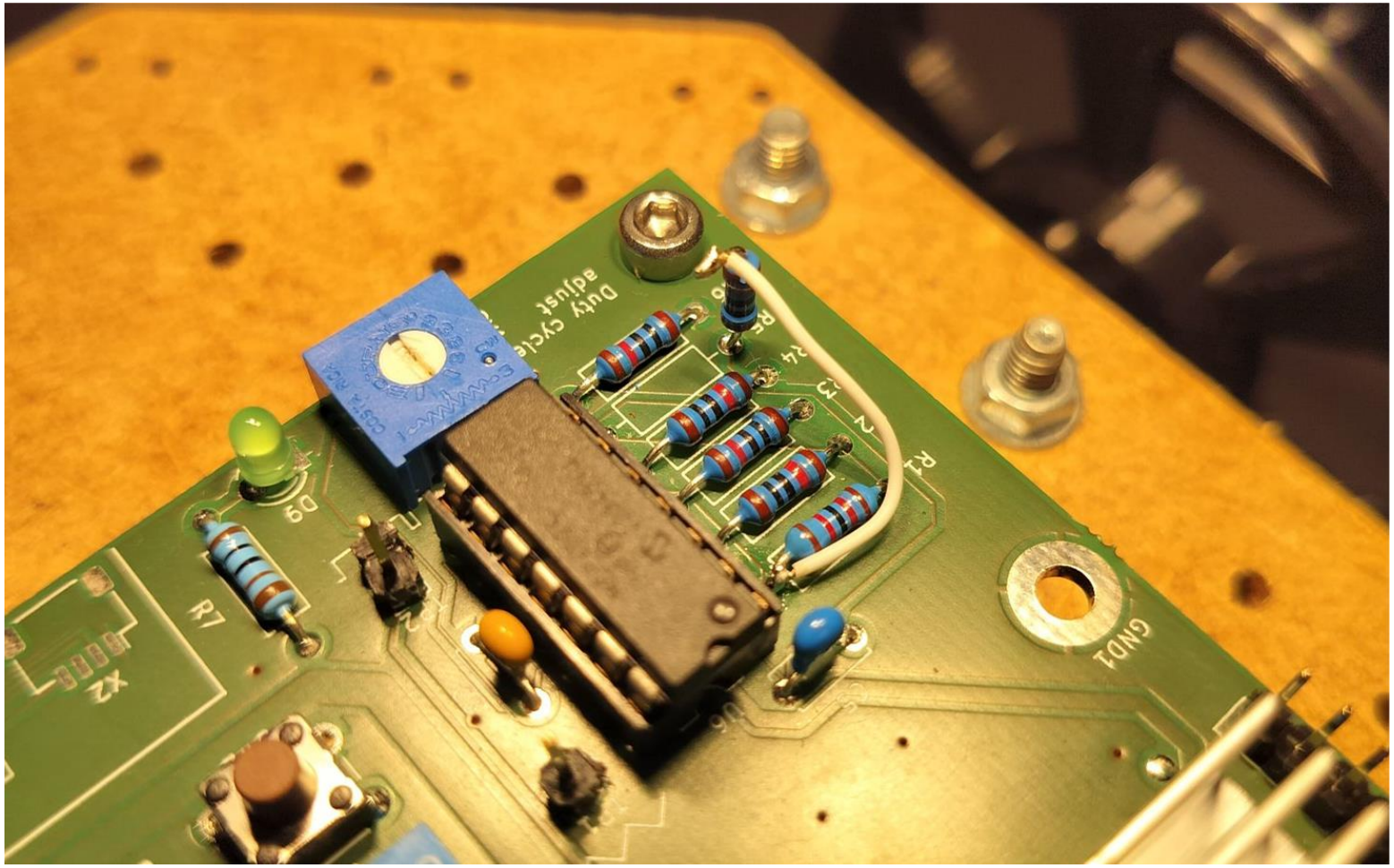


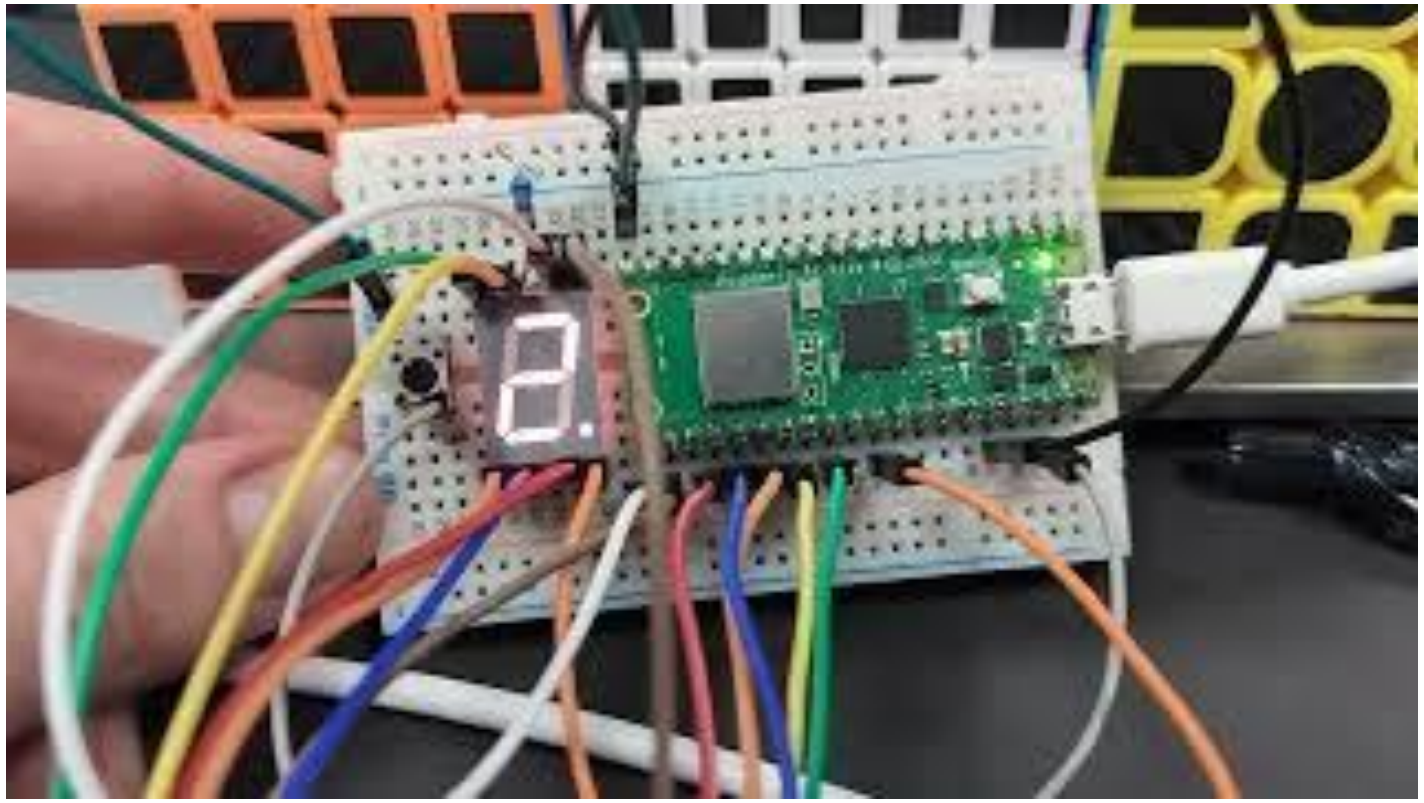
Illustration: https://anasounds.com/learn-how-to-solder-like-a-pro/

Practice PCB (Hardware fix)



Portfolio 1

Q/A?



Object-Oriented Programming (OOP) concepts

Object-Oriented Programming (OOP)

“...a programming paradigm based on the concept of “objects”, which can contain data and code”

Object-Oriented Programming (OOP)

"...a programming paradigm based on the concept of "objects", which can contain data and code"

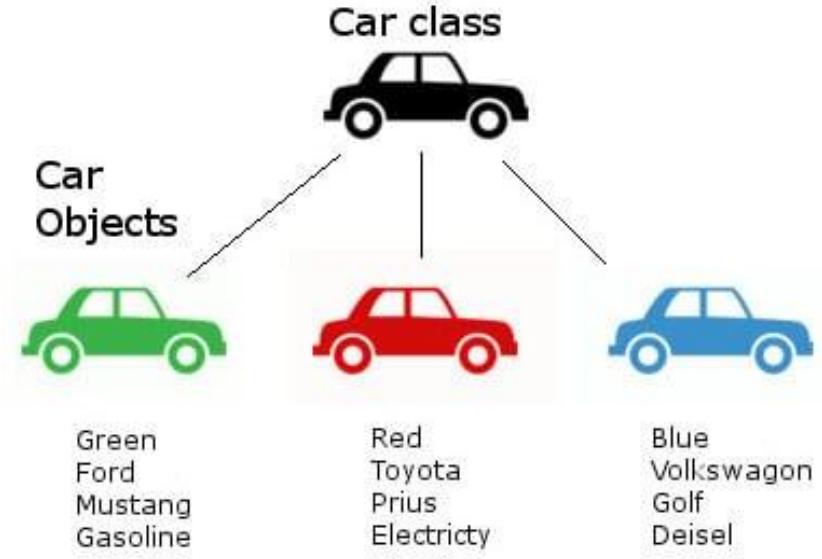
- **Data** in the form of fields (often known as attributes or properties), and
- **Code** in the form of procedures (often known as methods or functions)

Object-Oriented Programming (OOP)

"...a programming paradigm based on the concept of "objects", which can contain data and code"

- **Data** in the form of fields (often known as attributes or properties), and
- **Code** in the form of procedures (often known as methods or functions)

- Like a car factory
  - Produces multiple cars (Objects)
  - Each car might look the same
    - but with different attributes (color,engine, etc.)



- **Programming languages** that are known for their OOP capabilities:
  - C++, Java, Python, C#

Illustration: https://medium.com/pongpichs/oop-object-oriented-programing-4dda39dbb745

Object-Oriented Programming (OOP)

"...a programming paradigm based on the concept of "objects", which can contain data and code"

- **Why?**

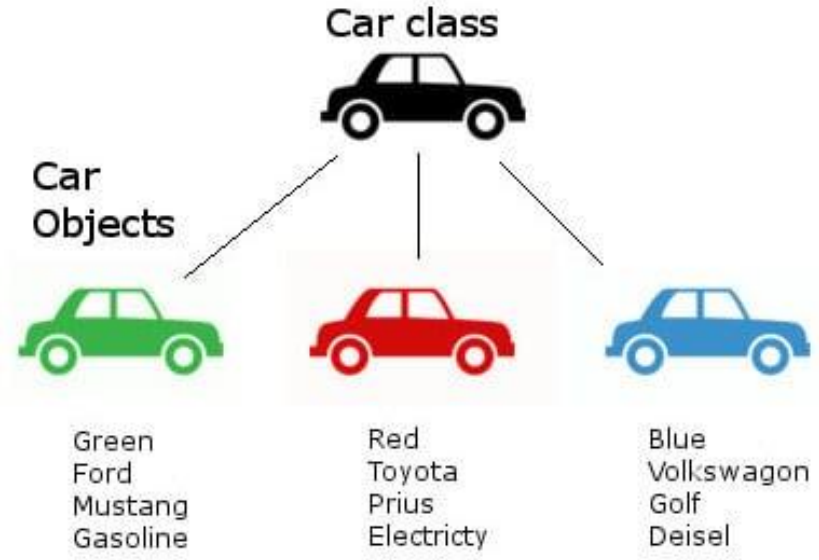


Illustration: https://medium.com/pongpichs/oop-object-oriented-programing-4dda39dbb745

Object-Oriented Programming (OOP)

"...a programming paradigm based on the concept of "objects", which can contain data and code"

- **Why?**
  - **Modularity**: An object can be written and maintained independently of the source code for other objects.
    - They can be removed, added, or replaced without affecting the rest of the program (to much…).

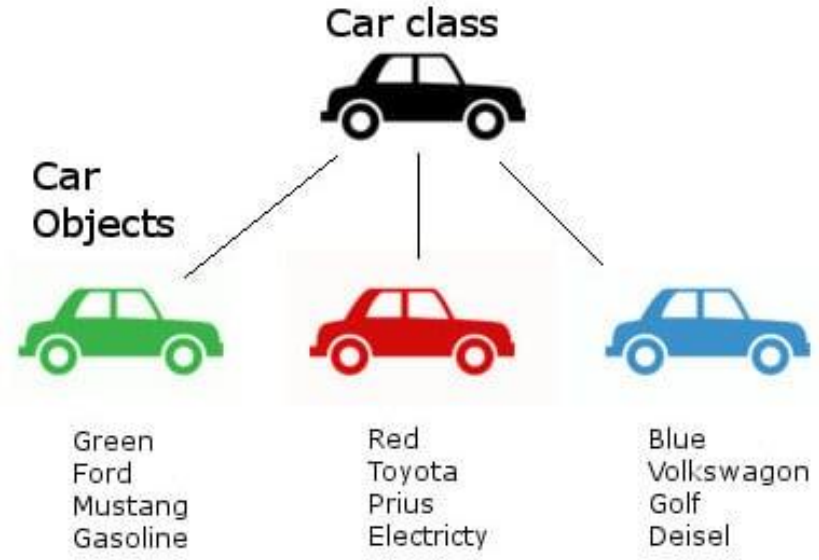


Illustration: https://medium.com/pongpichs/oop-object-oriented-programing-4dda39dbb745

Object-Oriented Programming (OOP)

"...a programming paradigm based on the concept of "objects", which can contain data and code"

- **Why?**
  - **Modularity**: An object can be written and maintained independently of the source code for other objects.
    - They can be removed, added, or replaced without affecting the rest of the program (to much…).
  - **Reusability / Distribution**: Objects can be reused across programs and can easily be shared.

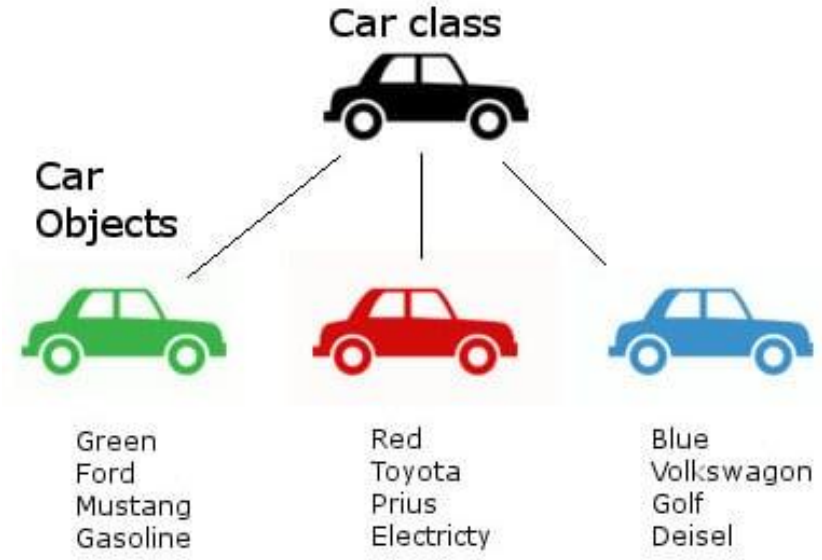


Illustration: https://medium.com/pongpichs/oop-object-oriented-programing-4dda39dbb745

Object-Oriented Programming (OOP)

"...a programming paradigm based on the concept of "objects", which can contain data and code"

- **Why?**
  - **Modularity**: An object can be written and maintained independently of the source code for other objects.
    - They can be removed, added, or replaced without affecting the rest of the program (to much…).
  - **Reusability / Distribution**: Objects can be reused across programs and can easily be shared.
  - **Scalability and Manageability**: OOP helps in managing and scaling large software projects by breaking them down into smaller, manageable objects.
    - If they object is well defined, the task of developing can be assigned to each group member or outsourced.

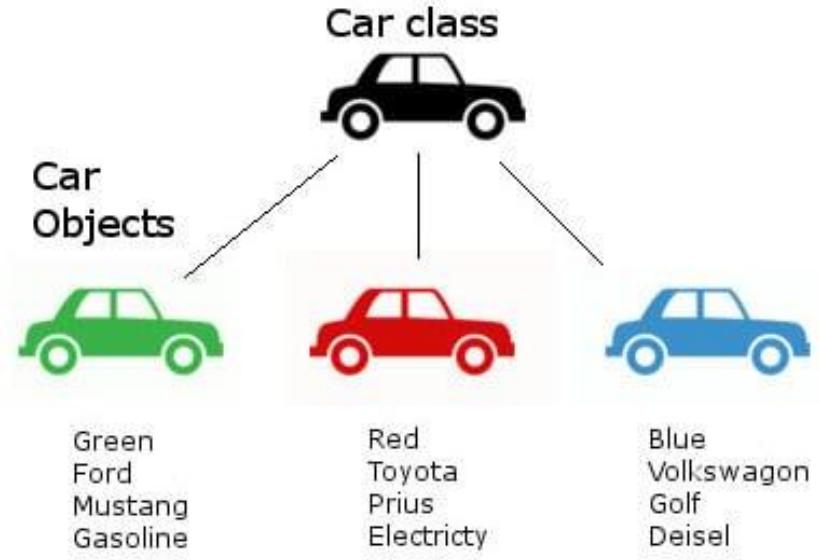


Illustration: https://medium.com/pongpichs/oop-object-oriented-programing-4dda39dbb745

Object-Oriented Programming (OOP)

"...a programming paradigm based on the concept of objects, which can contain data and code"

- **Key concepts**

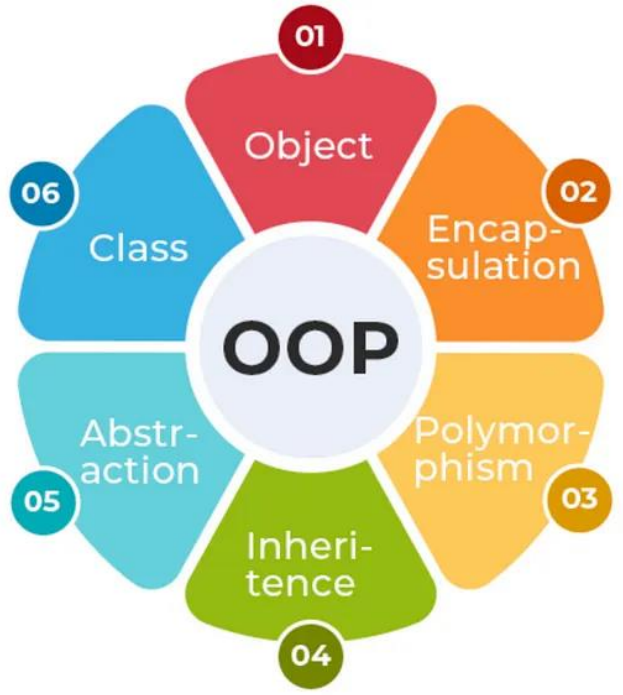


Illustration: https://medium.com/@sounder.rahul/python-oops-concepts-for-data-engineer-beginners-0b49d0fbce3e

Object-Oriented Programming (OOP)

"...a programming paradigm based on the concept of objects, which can contain data and code"

- **Key concepts**
  - **Classes and Objects:** A class is a template (or blueprint) for creating objects.
    - It defines a set of attributes and methods that the created objects will have. An object is an instance of a class.

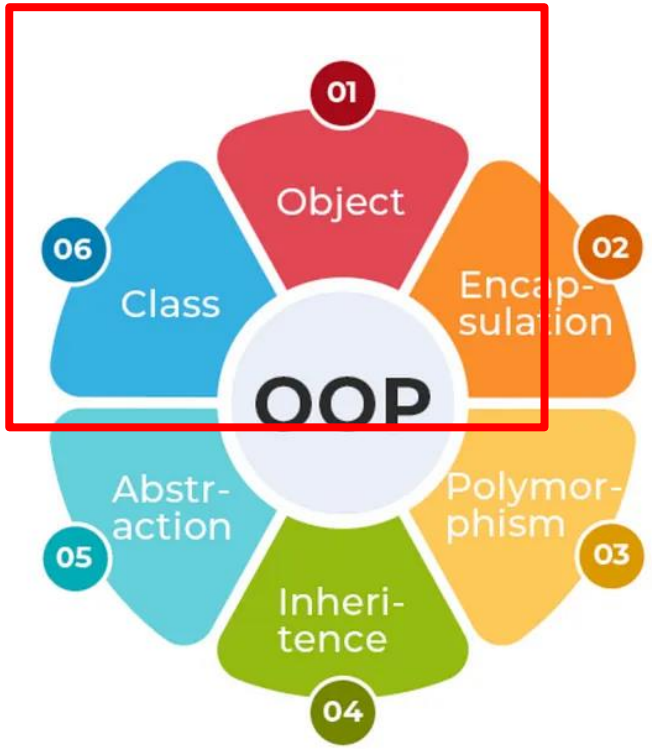


Illustration: https://medium.com/@sounder.rahul/python-oops-concepts-for-data-engineer-beginners-0b49d0fbce3e

Object-Oriented Programming (OOP)

"...a programming paradigm based on the concept of objects, which can contain data and code"

- **Key concepts**
  - **Classes and Objects:** A class is a template (or blueprint) for creating objects.
    - It defines a set of attributes and methods that the created objects will have. An object is an instance of a class.
  - **Encapsulation:** This principle involves bundling the data (attributes) and procedures (methods/functions) that operate on the data into a single unit or class.
    - Encapsulation helps in keeping the internal state of an object hidden from the outside, exposing only what is necessary.

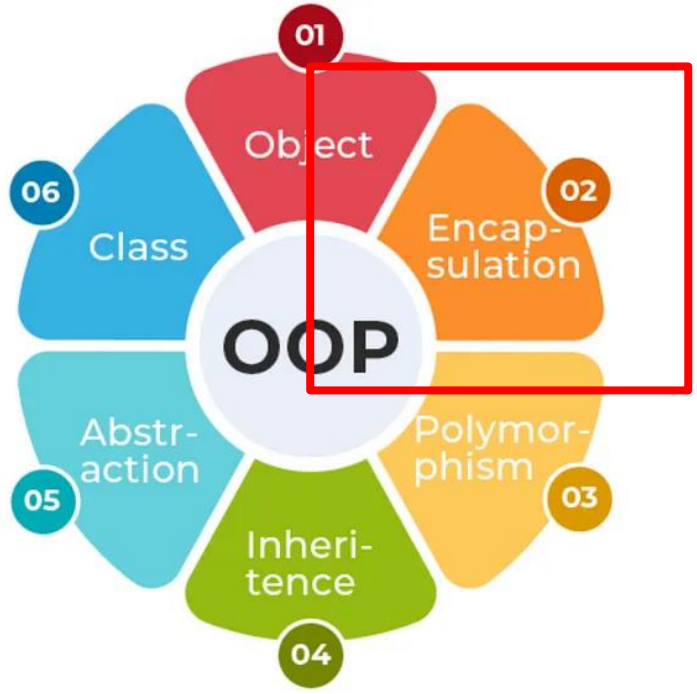


Illustration: https://medium.com/@sounder.rahul/python-oops-concepts-for-data-engineer-beginners-0b49d0fbce3e

Object-Oriented Programming (OOP)

"...a programming paradigm based on the concept of objects, which can contain data and code"

- **Key concepts**
  - **Classes and Objects:** A class is a template (or blueprint) for creating objects.
    - It defines a set of attributes and methods that the created objects will have. An object is an instance of a class.
  - **Encapsulation:** This principle involves bundling the data (attributes) and procedures (methods/functions) that operate on the data into a single unit or class.
    - Encapsulation helps in keeping the internal state of an object hidden from the outside, exposing only what is necessary.
  - **Inheritance:** Inheritance allows one class (child class) to inherit attributes and methods from another class (parent class).
    - This promotes code reuse and establishes a natural hierarchy between classes.

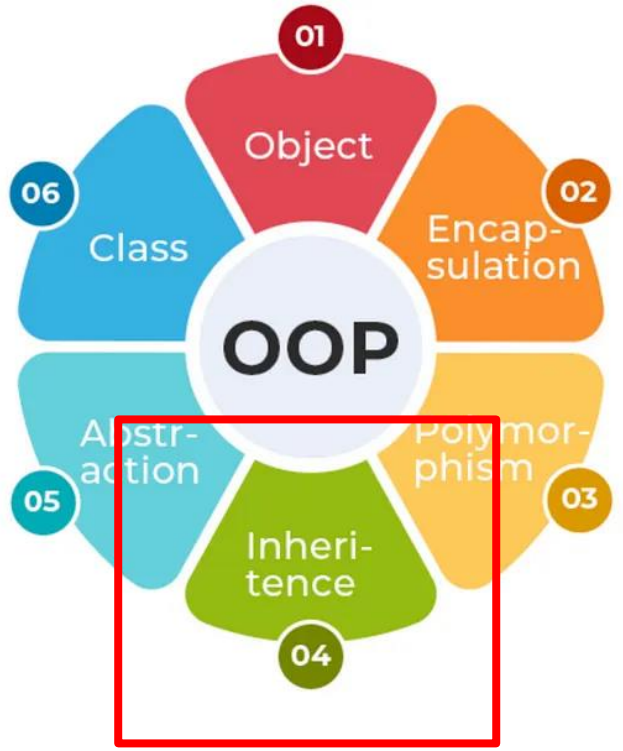


Illustration: https://medium.com/@sounder.rahul/python-oops-concepts-for-data-engineer-beginners-0b49d0fbce3e

Object-Oriented Programming (OOP)

"...a programming paradigm based on the concept of objects, which can contain data and code"

- **Key concepts**
  - **Classes and Objects:** A class is a template (or blueprint) for creating objects.
    - It defines a set of attributes and methods that the created objects will have. An object is an instance of a class.
  - **Encapsulation:** This principle involves bundling the data (attributes) and procedures (methods/functions) that operate on the data into a single unit or class.
    - Encapsulation helps in keeping the internal state of an object hidden from the outside, exposing only what is necessary.
  - **Inheritance:** Inheritance allows one class (child class) to inherit attributes and methods from another class (parent class).
    - This promotes code reuse and establishes a natural hierarchy between classes.
  - **Polymorphism:** Polymorphism enables objects of different classes to be treated as objects of a common super class.
    - This is often implemented through method overriding, where a subclass provides a specific implementation of a method that is already defined in its superclass.

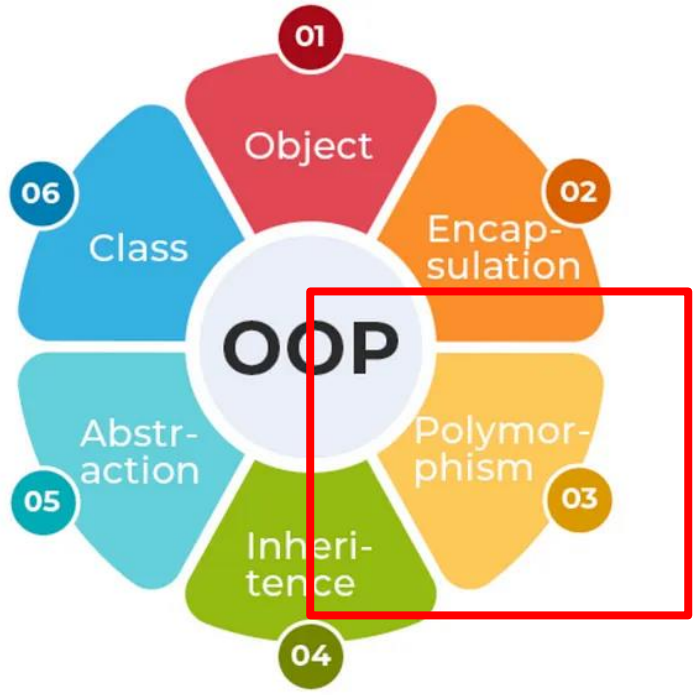


Illustration: https://medium.com/@sounder.rahul/python-oops-concepts-for-data-engineer-beginners-0b49d0fbce3e

Object-Oriented Programming (OOP)

"...a programming paradigm based on the concept of objects, which can contain data and code"

- **Key concepts**
  - **Classes and Objects:** A class is a template (or blueprint) for creating objects.
    - It defines a set of attributes and methods that the created objects will have. An object is an instance of a class.
  - **Encapsulation:** This principle involves bundling the data (attributes) and procedures (methods/functions) that operate on the data into a single unit or class.
    - Encapsulation helps in keeping the internal state of an object hidden from the outside, exposing only what is necessary.
  - **Inheritance:** Inheritance allows one class (child class) to inherit attributes and methods from another class (parent class).
    - This promotes code reuse and establishes a natural hierarchy between classes.
  - **Polymorphism:** Polymorphism enables objects of different classes to be treated as objects of a common super class.
    - This is often implemented through method overriding, where a subclass provides a specific implementation of a method that is already defined in its superclass.
  - **Abstraction:** The concept of hiding complex implementation details and showing only the essential features of an object.
    - It simplifies interaction with the object by providing a clear and simple interface.

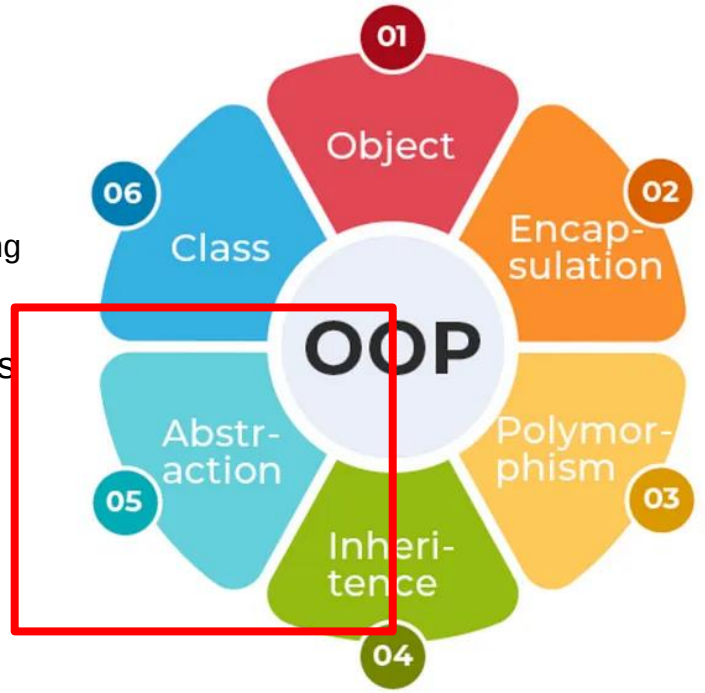


Illustration: https://medium.com/@sounder.rahul/python-oops-concepts-for-data-engineer-beginners-0b49d0fbce3e

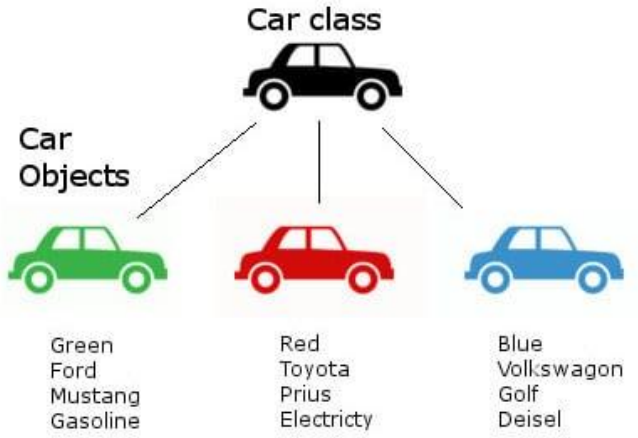
Classes and Objects

"...a template for creating objects."

In this example:

- The `Car` class defines
  a. Three **attributes** (`brand`, `model`, `year`) and
  b. a **method** (`start_engine`).

- The `my_car` object is an instance of the `Car` class, with specific values for its attributes.



Example: Define a Class and make an Object of the class

```
class Car:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year

    def start_engine(self):
        return f"The {self.year} {self.brand} {self.model}'s engine starts."

# Creating an object
my_car = Car("Toyota", "Corolla", 2020)

# Accessing attributes and methods
print(my_car.brand)           # Output: Toyota
print(my_car.start_engine()) # Output: The 2020 Toyota Corolla's engine starts.
```

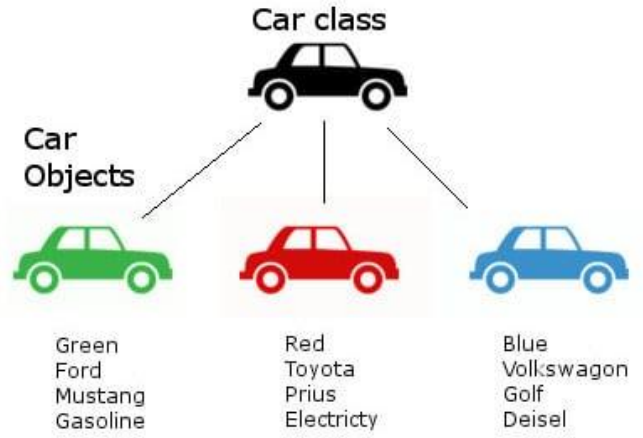

Illustration: https://medium.com/pongpichs/oop-object-oriented-programing-4dda39dbb745

Classes and Objects

"...a template for creating objects."

In this example:

- The `Car` class defines three attributes (`brand`, `model`, `year`) and a method (`start_engine`).
- The `my_car` object is an instance of the `Car` class, with specific values for its attributes.
- **Key concepts**



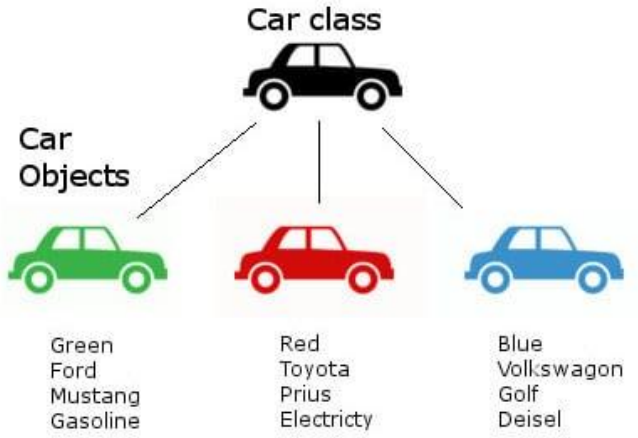
Example: Define a Class and make an Object of the class

```
class Car:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year

    def start_engine(self):
        return f"The {self.year} {self.brand} {self.model}'s engine starts."

# Creating an object
my_car = Car("Toyota", "Corolla", 2020)

# Accessing attributes and methods
print(my_car.brand)           # Output: Toyota
print(my_car.start_engine()) # Output: The 2020 Toyota Corolla's engine starts.
```


Illustration: https://medium.com/pongpichs/oop-object-oriented-programing-4dda39dbb745

Classes and Objects

"...a template for creating objects."



In this example:

- The `Car` class defines three attributes (`brand`, `model`, `year`) and a method (`start_engine`).
- The `my_car` object is an instance of the `Car` class, with specific values for its attributes.
- **Key concepts**
  a. **Constructor** (or **`__init__`** method)**:** When an instance of the object is created, the `__init__` method is automatically called.
     i. It take the arguments as inputs
     ii. …typically used for initializing variable
     iii. Input to the Car class:
        1. `make`, `model`, `year` (no defaults)
        2. However, `__init__` takes four?

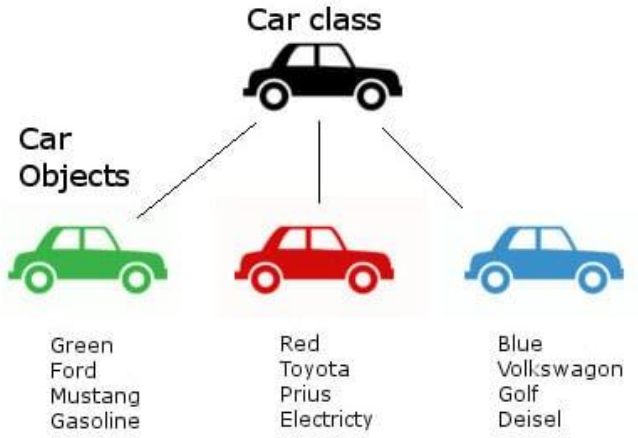
Example: Define a Class and make an Object of the class

```
class Car:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year

    def start_engine(self):
        return f"The {self.year} {self.brand} {self.model}'s engine starts."

# Creating an object
my_car = Car("Toyota", "Corolla", 2020)

# Accessing attributes and methods
print(my_car.brand)           # Output: Toyota
print(my_car.start_engine()) # Output: The 2020 Toyota Corolla's engine starts.
```


Illustration: https://medium.com/pongpichs/oop-object-oriented-programing-4dda39dbb745

Classes and Objects

"...a template for creating objects."



In this example:

- The `Car` class defines three attributes (`brand`, `model`, `year`) and a method (`start_engine`).
- The `my_car` object is an instance of the `Car` class, with specific values for its attributes.
- **Key concepts**
  a. **Constructor** (or **`__init__`** method)**:** When an instance of the object is created, the `__init__` method is automatically called.
     i. It take the arguments as inputs
     ii. …typically used for initializing variable
     iii. Input to the Car class:
        1. `make`, `model`, `year` (no defaults)
        2. However, `__init__` takes four?

  b. **`self` Parameter:** a reference to the current instance of the class, and is used to access variables that belongs to the class.

Example: Define a Class and make an Object of the class

```
class Car:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year

    def start_engine(self):
        return f"The {self.year} {self.brand} {self.model}'s engine starts."

# Creating an object
my_car = Car("Toyota", "Corolla", 2020)

# Accessing attributes and methods
print(my_car.brand)           # Output: Toyota
print(my_car.start_engine()) # Output: The 2020 Toyota Corolla's engine starts.
```


Illustration: https://medium.com/pongpichs/oop-object-oriented-programing-4dda39dbb745

Classes and Objects

"...a template for creating objects."

Organize: Same example but in separate files:

- The `Car` class is defined in it own file (`garage.py`).
- The `main.py` file imports the `Car` class from the `garage` file.

= Simplifies the main file

= **(Modularity)** The class can be imported by multiple files at the same time

Example: Split up into separate files

File: garage.py

```
class Car:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year

    def start_engine(self):
        return f"The {self.year} {self.brand} {self.model}'s engine starts."
```

File: main.py

```
from garage import Car # Import the Car class from Garage.py

# Creating an object
my_car = Car("Toyota", "Corolla", 2020)

# Accessing attributes and methods
print(my_car.brand)            # Output: Toyota
print(my_car.start_engine()) # Output: The 2020 Toyota Corolla's engine starts.
```

File structure (on the Pico)

"...organizing files and directories (folders) in a logical and modular way to make code more manageable, readable, and maintainable."

In this example:

- **First,** put your class definitions in separate Python files (modules).
  a. For example, you have a class `Car` that you want to use in multiple files.
  b. Create a file named `garage.py` and put your class there.

- Then, in the main script (or another file where you want to use the class):
  a. import it using the `import` statement.

- **Ensure both files are in the same directory,** or
- **handle imports by specifying relative or absolute paths.**

Example: Split up into separate files

File: garage.py

```
class Car:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year

    def start_engine(self):
        return f"The {self.year} {self.brand} {self.model}'s engine starts."
```

File: main.py

```
from garage import Car # Import the Car class from Garage.py

# Creating an object
my_car = Car("Toyota", "Corolla", 2020)

# Accessing attributes and methods
print(my_car.brand)            # Output: Toyota
print(my_car.start_engine()) # Output: The 2020 Toyota Corolla's engine starts.
```

File structure (on the Pico)

"...organizing files and directories (folders) in a logical and modular way to make code more manageable, readable, and maintainable."

In this example:

- **First,** put your class definitions in separate Python files (modules).
  a. For example, you have a class `Car` that you want to use in multiple files.
  b. Create a file named `garage.py` and put your class there.

- Then, in the main script (or another file where you want to use the class):
  a. import it using the `import` statement.

- **Ensure both files are in the same directory,** or
- **handle imports by specifying relative or absolute paths.**

Example using Thonny

Example: Split up into separate files

File: garage.py

```
class Car:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year

    def start_engine(self):
        return f"The {self.year} {self.brand} {self.model}'s engine starts."
```

File: main.py

```
from garage import Car # Import the Car class from Garage.py

# Creating an object
my_car = Car("Toyota", "Corolla", 2020)

# Accessing attributes and methods
print(my_car.brand)            # Output: Toyota
print(my_car.start_engine()) # Output: The 2020 Toyota Corolla's engine starts.
```

Encapsulation

"...bundling the data (attributes) and procedures (methods/functions) that operate on the data into a single unit or class."

In this example:

- Created a class for managing a bank account

Example: Define a Class and make an Object of the class

```
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner
        self.__balance = balance  # Private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            return f"${amount} deposited. New balance: ${self.__balance}"
        return "Deposit amount must be positive."

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            return f"${amount} withdrawn. New balance: ${self.__balance}"
        return "Insufficient balance or invalid amount."

    def get_balance(self):
        return f"Balance: ${self.__balance}"

# Creating an object
account = BankAccount("Alice", 1000)
```

Encapsulation

"...bundling the data (attributes) and procedures (methods/functions) that operate on the data into a single unit or class."

In this example:

- Created a class for managing a bank account
- The `__balance` attribute is private and cannot be accessed directly from outside the class.
- Methods like `deposit`, `withdraw`, and `get_balance` provide controlled access to the `__balance` attribute.

Example: Define a Class and make an Object of the class

```
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner
        self.__balance = balance  # Private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            return f"${amount} deposited. New balance: ${self.__balance}"
        return "Deposit amount must be positive."

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            return f"${amount} withdrawn. New balance: ${self.__balance}"
        return "Insufficient balance or invalid amount."

    def get_balance(self):
        return f"Balance: ${self.__balance}"

# Creating an object
account = BankAccount("Alice", 1000)

# Using methods to interact with the balance
print(account.deposit(500))      # Output: $500 deposited. New balance: $1500
print(account.withdraw(200))     # Output: $200 withdrawn. New balance: $1300
print(account.get_balance())     # Output: Balance: $1300
# print(account.__balance)       # This would raise an AttributeError
```

Inheritance

“...allows one class (child class) to inherit attributes and methods from another class (parent class).”

In this example:

- Created a class for a generic animal

Example: Define a Class and make an Object of the class

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"{self.name} makes a sound."
```

Inheritance

“...allows one class (child class) to inherit attributes and methods from another class (parent class).”

In this example:

- Created a class for a generic animal
- `Dog` and `Cat` classes inherit from the `Animal` class.
- They override the `speak` method to provide behavior specific to dogs and cats.

Example: Define a Class and make an Object of the class

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"{self.name} makes a sound."

class Dog(Animal):  # Dog inherits from Animal
    def speak(self):
        return f"{self.name} barks."

class Cat(Animal):  # Cat inherits from Animal
    def speak(self):
        return f"{self.name} meows."

# Creating objects
dog = Dog("Buddy")
cat = Cat("Whiskers")

# Using inherited and overridden methods
print(dog.speak())  # Output: Buddy barks.
print(cat.speak())  # Output: Whiskers meows.
```

Polymorphism

“...enables objects of different classes to be treated as objects of a common super class.”

= "having multiple forms."

In this example:

- Created a class for a generic bird, and inherit from it for two other classes.

Example: Define a Class and make an Object of the class

```
class Bird:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"{self.name} chirps."

class Parrot(Bird):
    def speak(self):
        return f"{self.name} talks."

class Crow(Bird):
    def speak(self):
        return f"{self.name} caws."
```

Polymorphism

“...enables objects of different classes to be treated as objects of a common super class.”

= "having multiple forms."

In this example:

- Created a class for a generic bird, and inherit from it for two other classes.
- The `make_bird_speak` function can take any object of a class that inherits from `Bird` and call the `speak` method, demonstrating polymorphism.

Example: Define a Class and make an Object of the class

```
class Bird:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"{self.name} chirps."

class Parrot(Bird):
    def speak(self):
        return f"{self.name} talks."

class Crow(Bird):
    def speak(self):
        return f"{self.name} caws."

# Function that takes a Bird object and calls its speak method
def make_bird_speak(bird):
    print(bird.speak())

# Creating objects
parrot = Parrot("Polly")
crow = Crow("Blackie")

# Using polymorphism
make_bird_speak(parrot)   # Output: Polly talks.
make_bird_speak(crow)     # Output: Blackie caws.
```

Abstraction

"...the concept of hiding complex implementation details and showing only the essential features of an object."

In this example:

- `Shape` is an abstract class with an abstract method `area`.
  a. Simplify what is relevant
  b. Define what is needed for the Class, and "hide" the rest
- `Rectangle` is a concrete class that implements the `area` method.
- The user interacts with the `Rectangle` object through the `area` method without needing to know how the area is calculated internally.

Example: Define a Class and make an Object of the class

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

# Creating an object and using the abstracted method
rect = Rectangle(5, 3)
print(rect.area())  # Output: 15
```

Actuators

Actuators

“Components that are responsible for moving or controlling the robot's mechanical systems. They convert electrical signals (energy) from the robot's control system into physical motion”

Roles of Actuators in Mobile Robots:

Illustrations: from Google Images

Actuators

“Components that are responsible for moving or controlling the robot's mechanical systems. They convert electrical signals (energy) from the robot's control system into physical motion”

Roles of Actuators in Mobile Robots:

- **Locomotion**: Moving the robot from one place to another, either through wheels, tracks, legs, or other means.

Illustrations: from Google Images

Actuators

“Components that are responsible for moving or controlling the robot's mechanical systems. They convert electrical signals (energy) from the robot's control system into physical motion”

Roles of Actuators in Mobile Robots:

- **Locomotion**: Moving the robot from one place to another, either through wheels, tracks, legs, or other means.
- **Manipulation**: Controlling robotic arms or grippers to pick up, move, or manipulate objects.

Illustrations: from Google Images

Actuators

“Components that are responsible for moving or controlling the robot's mechanical systems. They convert electrical signals (energy) from the robot's control system into physical motion”

Roles of Actuators in Mobile Robots:

- **Locomotion**: Moving the robot from one place to another, either through wheels, tracks, legs, or other means.
- **Manipulation**: Controlling robotic arms or grippers to pick up, move, or manipulate objects.
- **Steering**: Controlling the direction in which the robot moves.

Illustrations: from Google Images

Actuators

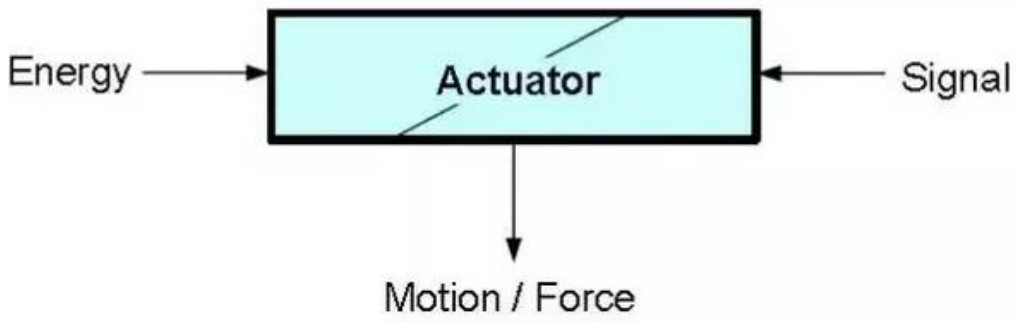
“Components that are responsible for moving or controlling the robot's mechanical systems. They convert electrical signals (energy) from the robot's control system into physical motion”

Roles of Actuators in Mobile Robots:

- **Locomotion**: Moving the robot from one place to another, either through wheels, tracks, legs, or other means.
- **Manipulation**: Controlling robotic arms or grippers to pick up, move, or manipulate objects.
- **Steering**: Controlling the direction in which the robot moves.

(...so essentially, the ***"muscles"*** of a robot, allowing it to move, manipulate objects, and interact with its environment.)

Illustrations: from Google Images

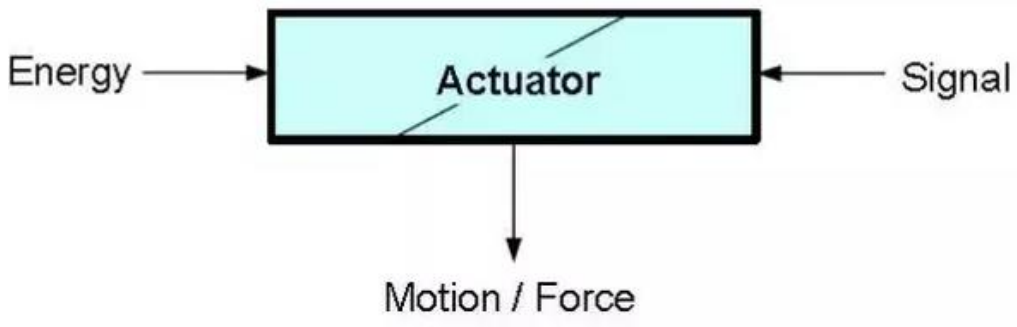


Actuators

"Components that are responsible for moving or controlling the robot's mechanical systems. They convert electrical signals (energy) from the robot's control system into physical motion"

Types of Actuators (typical source of energy)

Illustrations: https://www.quora.com/What-is-electric-actuator and Google Images



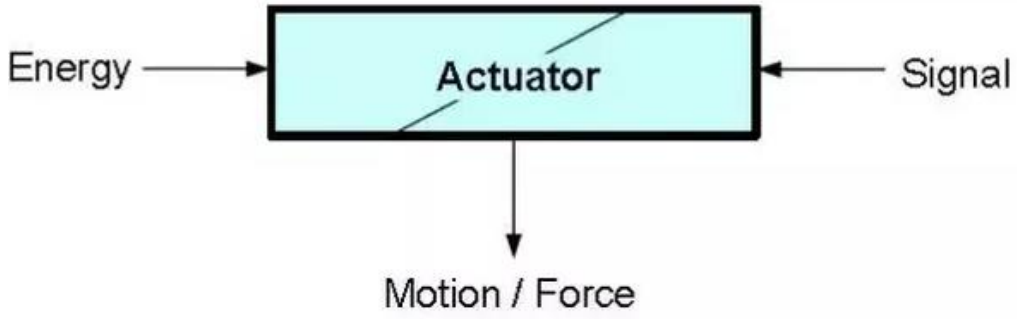
Actuators

"Components that are responsible for moving or controlling the robot's mechanical systems. They convert electrical signals (energy) from the robot's control system into physical motion"

Types of Actuators (typical source of energy)

- **Pneumatic Actuators:** Use compressed air to generate movement, often used in robots where rapid motion is required.

Illustrations: https://www.quora.com/What-is-electric-actuator and Google Images

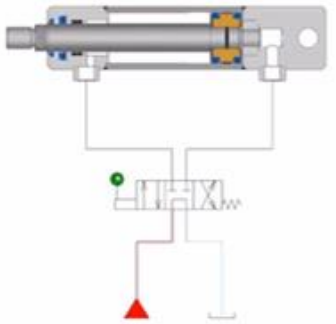


Actuators

"Components that are responsible for moving or controlling the robot's mechanical systems. They convert electrical signals (energy) from the robot's control system into physical motion"

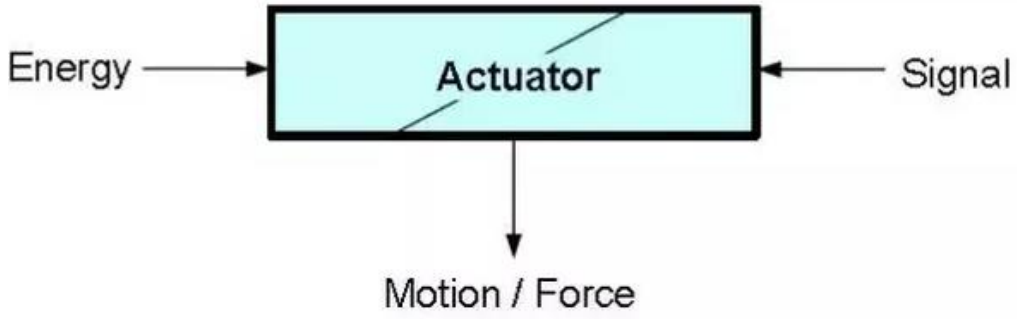
Types of Actuators (typical source of energy)

- **Pneumatic Actuators:** Use compressed air to generate movement, often used in robots where rapid motion is required.



- **Hydraulic Actuators:** Use pressurized fluid to create strong forces, suitable for heavy-duty tasks.

Illustrations: https://www.quora.com/What-is-electric-actuator and Google Images

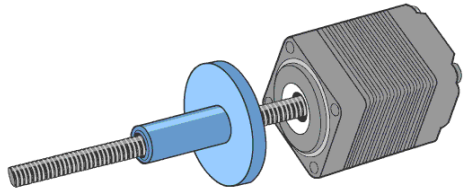


Actuators

"Components that are responsible for moving or controlling the robot's mechanical systems. They convert electrical signals (energy) from the robot's control system into physical motion"

Types of Actuators (typical source of energy)

- **Pneumatic Actuators:** Use compressed air to generate movement, often used in robots where rapid motion is required.
- **Hydraulic Actuators:** Use pressurized fluid to create strong forces, suitable for heavy-duty tasks.
- **Electric Actuators:** Use electrical energy to generate movement, typically using electromagnetism.



Illustrations: https://www.quora.com/What-is-electric-actuator and Google Images

Actuators

"Components that are responsible for moving or controlling the robot's mechanical systems. They convert electrical signals (energy) from the robot's control system into physical motion"

Electric Actuators (two types)

- **Rotary Actuators**: Produce rotational motion, meaning they rotate a component or a shaft around an axis.
- **Linear Actuators:** Produce straight-line motion, eg. by converting rotational motion into linear motion.

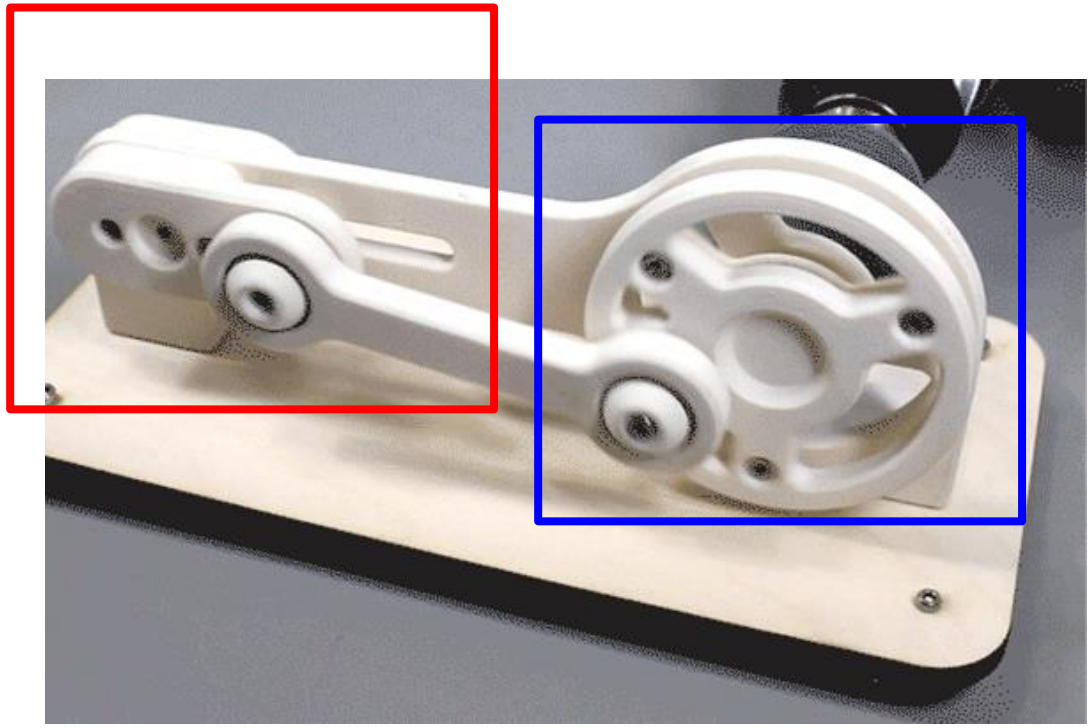


Illustration: https://ar.pinterest.com/pin/308707749472703218/

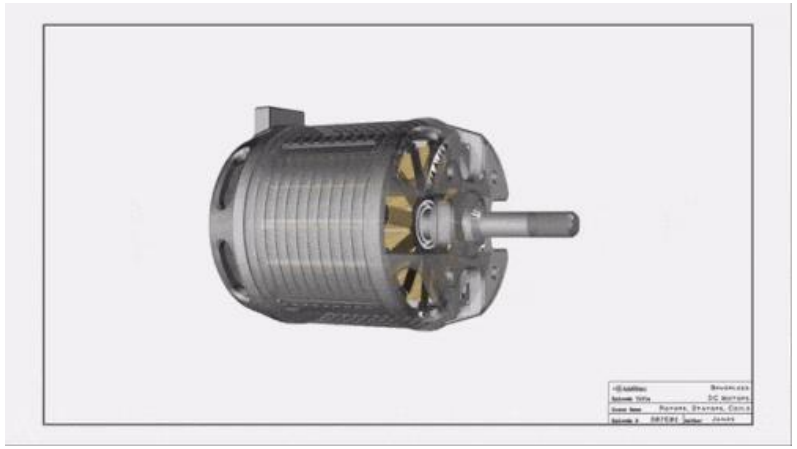
Actuators

“Components that are responsible for moving or controlling the robot's mechanical systems. They convert electrical signals (energy) from the robot's control system into physical motion”

Electric Actuators (Rotary Actuators)

Actuators

“Components that are responsible for moving or controlling the robot's mechanical systems. They convert electrical signals (energy) from the robot's control system into physical motion”



Electric Actuators (Rotary Actuators)

- **DC Motors:** Operate on direct current and provide continuous rotational motion.

Illustrations: from Google Images

Actuators

“Components that are responsible for moving or controlling the robot's mechanical systems. They convert electrical signals (energy) from the robot's control system into physical motion”

Electric Actuators (Rotary Actuators)

- **DC Motors:** Operate on direct current and provide continuous rotational motion.
- **Servo Motors:** A specialized type of motor that is designed for precise control of angular or linear position, velocity, and acceleration.

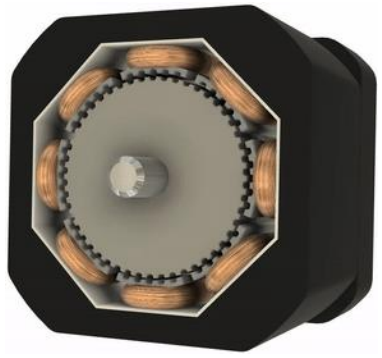
Illustrations: from Google Images

Actuators

“Components that are responsible for moving or controlling the robot's mechanical systems. They convert electrical signals (energy) from the robot's control system into physical motion”

Electric Actuators (Rotary Actuators)

- **DC Motors:** Operate on direct current and provide continuous rotational motion.
- **Servo Motors:** A specialized type of motor that is designed for precise control of angular or linear position, velocity, and acceleration.
- **Stepper Motors:** Move in discrete steps, allowing precise control over position, speed, and acceleration.

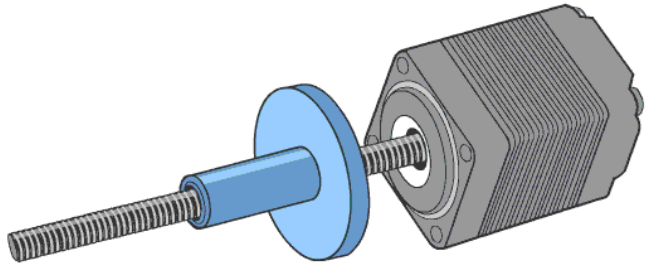


Illustrations: from Google Images

Actuators

“Components that are responsible for moving or controlling the robot's mechanical systems. They convert electrical signals (energy) from the robot's control system into physical motion”

Electric Actuators (Linear Actuators)



Actuators

“Components that are responsible for moving or controlling the robot's mechanical systems. They convert electrical signals (energy) from the robot's control system into physical motion”

Electric Actuators (Linear Actuators)

- **Screw Actuators (Lead or Ball):** Use a threaded screw mechanism where a motor turns the screw to move a nut along its length.

Illustrations: from Google Images

Actuators

“Components that are responsible for moving or controlling the robot's mechanical systems. They convert electrical signals (energy) from the robot's control system into physical motion”

Electric Actuators (Linear Actuators)

- **Screw Actuators (Lead or Ball):** Use a threaded screw mechanism where a motor turns the screw to move a nut along its length.

- **Belt-Driven Actuators:** Use a belt and pulley system for linear moti**on.**

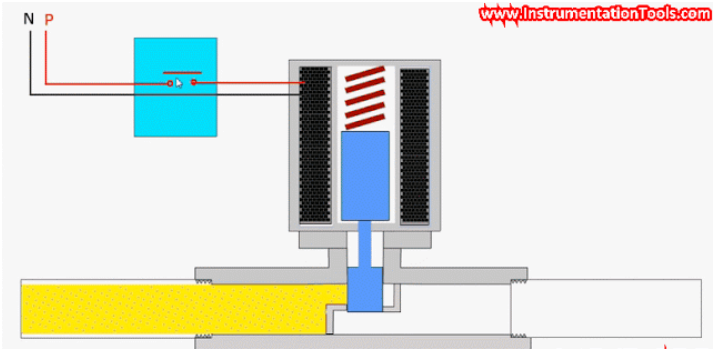
Illustrations: from Google Images

Actuators

“Components that are responsible for moving or controlling the robot's mechanical systems. They convert electrical signals (energy) from the robot's control system into physical motion”

Electric Actuators (Linear Actuators)

- **Screw Actuators (Lead or Ball):** Use a threaded screw mechanism where a motor turns the screw to move a nut along its length.
- **Belt-Driven Actuators:** Use a belt and pulley system for linear motion.
- **Solenoid Actuators:** Use electromagnetic force to move a plunger or armature in a linear direction.



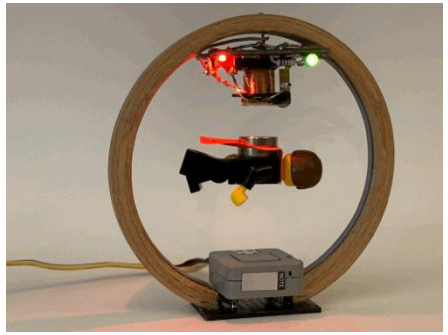
Illustrations: from Google Images

Actuators

“Components that are responsible for moving or controlling the robot's mechanical systems. They convert electrical signals (energy) from the robot's control system into physical motion”

Electric Actuators (Linear Actuators)

- **Screw Actuators (Lead or Ball):** Use a threaded screw mechanism where a motor turns the screw to move a nut along its length.
- **Belt-Driven Actuators:** Use a belt and pulley system for linear motion.
- **Solenoid Actuators:** Use electromagnetic force to move a plunger or armature in a linear direction.
- **Magnetic Linear Actuators:** Use magnetic fields to produce linear motion, often seen in magnetic levitation systems.



Illustrations: from Google Images

DC Motors

DC Motors

“Operates on Direct Current (DC) and provide continuous rotational motion (mechanical force)”

- **Control Methods:** DC motors are relatively easy to control, which makes them popular in robotics and automation, typically:
  - **Voltage Control:** The speed of a DC motor is directly proportional to the applied voltage.
    - = by varying the voltage, the speed of the motor can be adjusted.

DC Motors

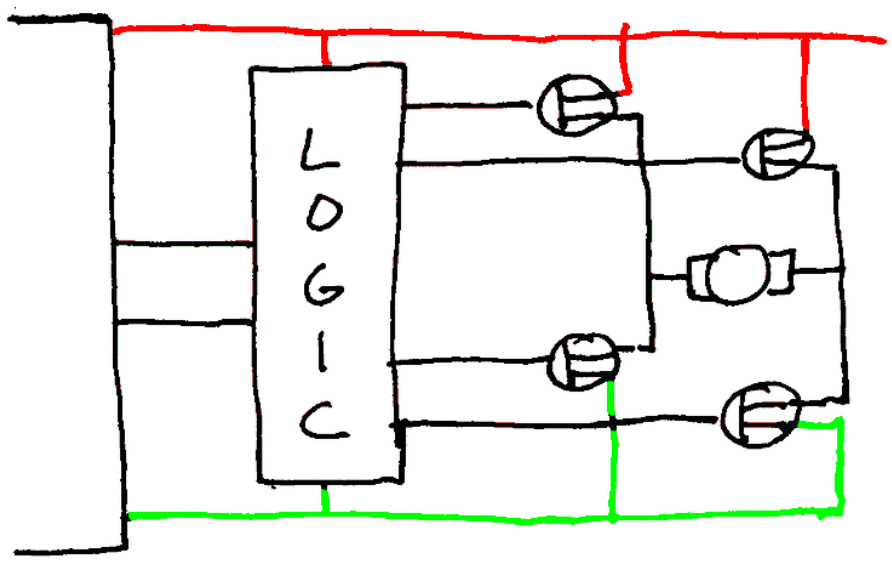
“Operates on Direct Current (DC) and provide continuous rotational motion (mechanical force)”

- **Control Methods:** DC motors are relatively easy to control, which makes them popular in robotics and automation, typically:
  - **Voltage Control:** The speed of a DC motor is directly proportional to the applied voltage.
    - = by varying the voltage, the speed of the motor can be adjusted.
  - **For most applications**, we want to be able to do two things with a motor:
    - Run it in forward and backward directions
    - Change its speed

DC Motors

"Operates on Direct Current (DC) and provide continuous rotational motion (mechanical force)"

- **Control Methods:** DC motors are relatively easy to control, which makes them popular in robotics and automation, typically:
  - **Voltage Control:** The speed of a DC motor is directly proportional to the applied voltage.
    - = by varying the voltage, the speed of the motor can be adjusted.
  - **For most applications**, we want to be able to do two things with a motor:
    - Run it in <u>forward</u> and <u>backward</u> directions
      - Using a **H-Bridge**
    - <u>Change its speed</u>



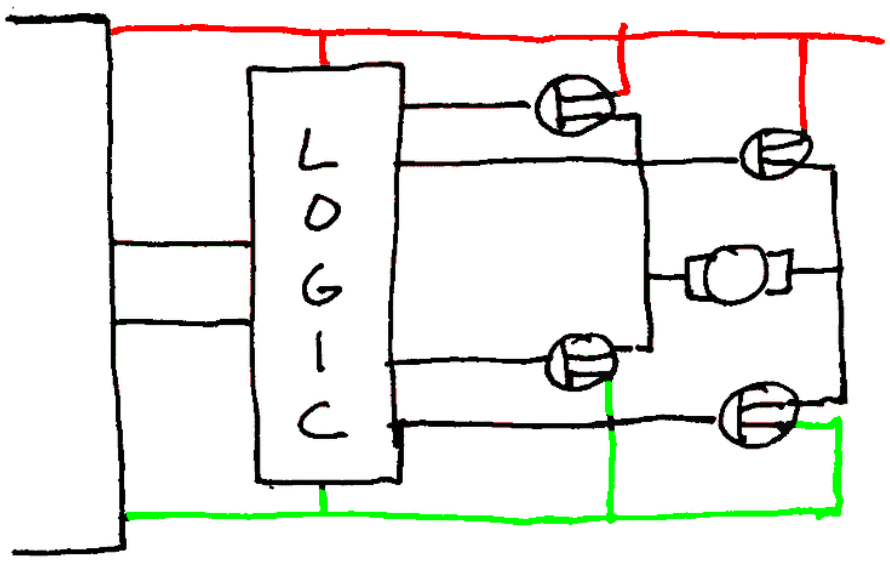
Microcontrollers are very limited output power

(eg. the **Pico has a maximum of 16mA per pin**)

(Typically) <u>Used to drive other logic chips</u>, but never a motor directly.

DC Motors

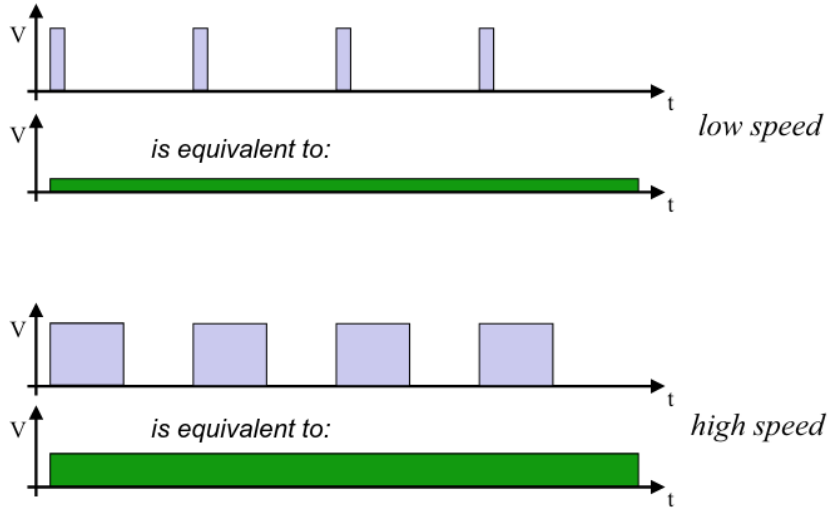
"Operates on Direct Current (DC) and provide continuous rotational motion (mechanical force)"



- **Control Methods:** DC motors are relatively easy to control, which makes them popular in robotics and automation, typically:
  - **Voltage Control:** The speed of a DC motor is directly proportional to the applied voltage.
    - = by varying the voltage, the speed of the motor can be adjusted.

  - **For most applications**, we want to be able to do two things with a motor:
    - Run it in <u>forward</u> and <u>backward</u> directions
      - Using a **H-Bridge**

    - <u>Change its speed</u>
      - Using **Pulse Width Modulation (PWM)**



H-Bridge

H-Bridge

“An electronic circuit that allows a voltage to be applied across a load, such as a DC motor, in either direction.”

= <u>change polarity</u> of the supply voltage on the motor…

- Gets its name from the typical arrangement of its four switching elements, which resemble the shape of the letter "H."

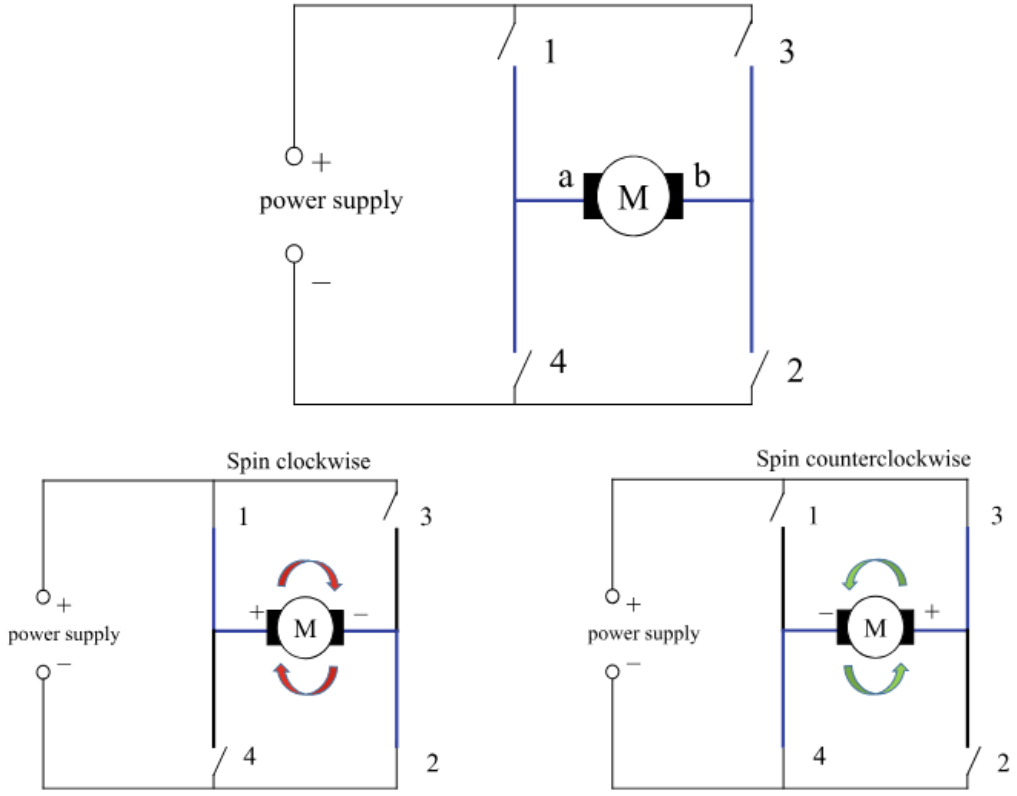


Illustration: [eB1] From Mobile Robots to Autonomous Vehicles with Raspberry Pi and Arduino

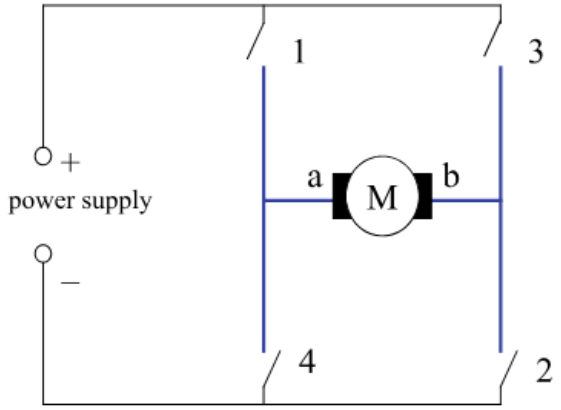
H-Bridge

"An electronic circuit that allows a voltage to be applied across a load, such as a DC motor, in either direction."

= <u>change polarity</u> of the supply voltage on the motor…

- Gets its name from the typical arrangement of its four switching elements, which resemble the shape of the letter "H."

- Basic Structure of an H-Bridge
  - **Four Switches:** The H-bridge has four switches
    - Eg. labeled S1, S2, S3, and S4.
  - **Load** (Motor)
  - **Power Supply**: provide the necessary voltage and current.



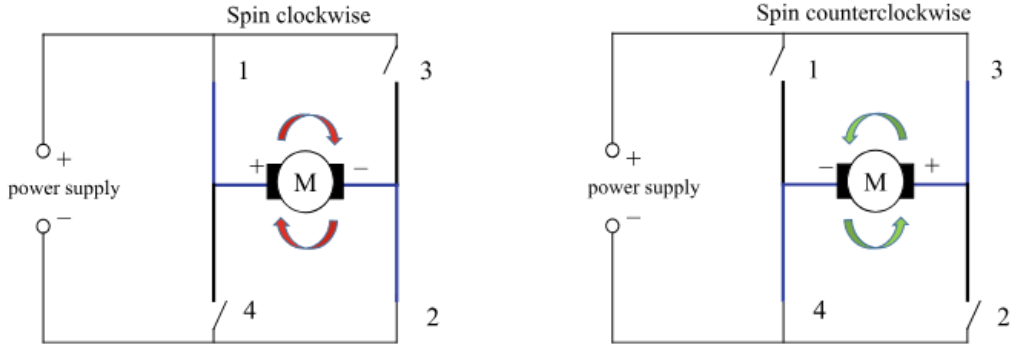


Illustration: [eB1] From Mobile Robots to Autonomous Vehicles with Raspberry Pi and Arduino

H-Bridge

"An electronic circuit that allows a voltage to be applied across a load, such as a DC motor, in either direction."

Controlling a H-Bridge

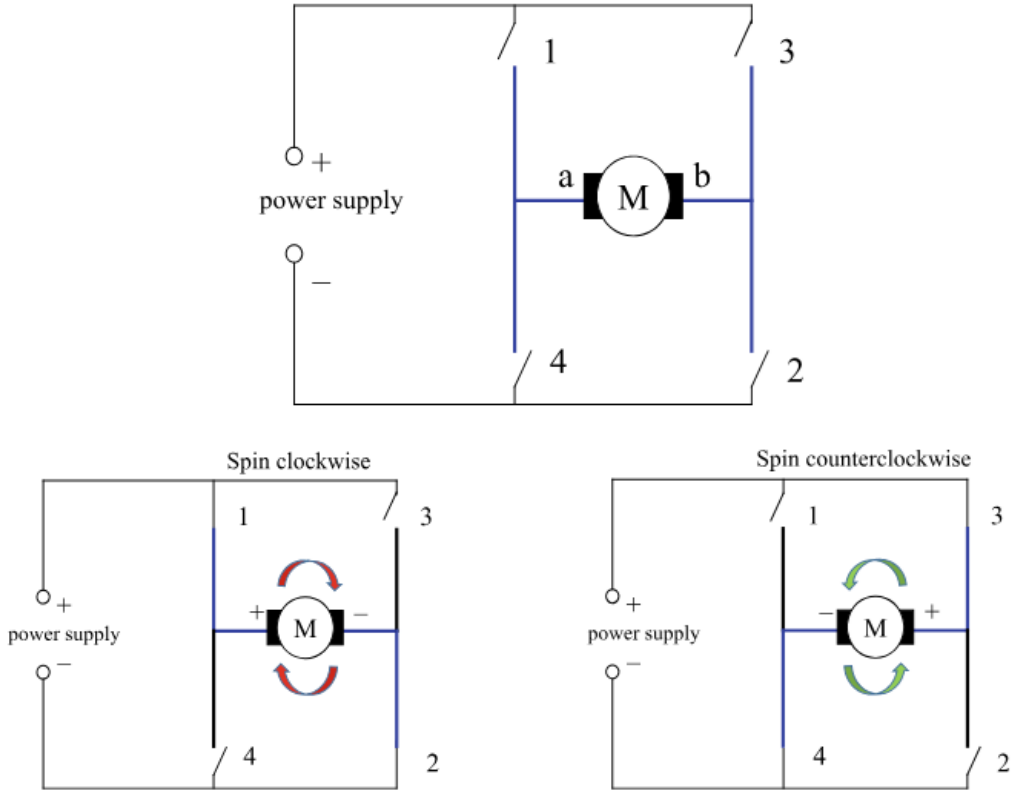


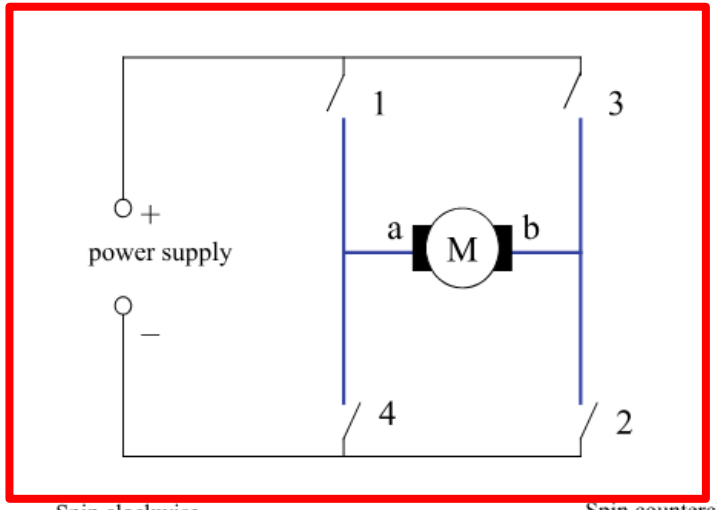
Illustration: [eB1] From Mobile Robots to Autonomous Vehicles with Raspberry Pi and Arduino

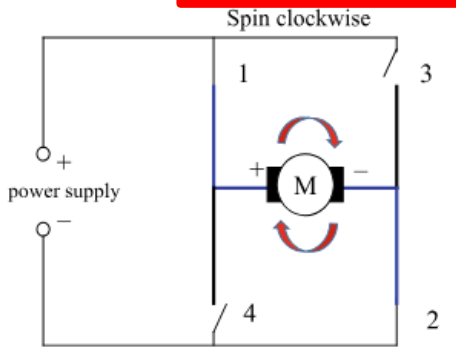
H-Bridge

"An electronic circuit that allows a voltage to be applied across a load, such as a DC motor, in either direction."

Controlling a H-Bridge

- **Coasting** (Motor Freewheeling)**:**
  - All switches open (1, 2, 3, and 4)
  - No current flows through the motor.





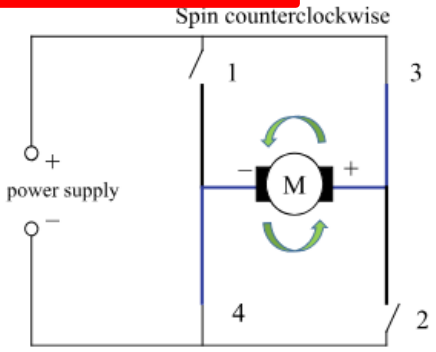


Illustration: [eB1] From Mobile Robots to Autonomous Vehicles with Raspberry Pi and Arduino

H-Bridge

"An electronic circuit that allows a voltage to be applied across a load, such as a DC motor, in either direction."

Controlling a H-Bridge

- **Coasting** (Motor Freewheeling)**:**
  - All switches open (1, 2, 3, and 4)
  - No current flows through the motor.

- **Forward Direction** (Clockwise Rotation)
  - **Switches Closed:** 1 and 2, **Switches Open:** 3 and 4
  - Current flows from the power supply, through 1, across the motor (from one terminal to the other), and then through 2 to ground.

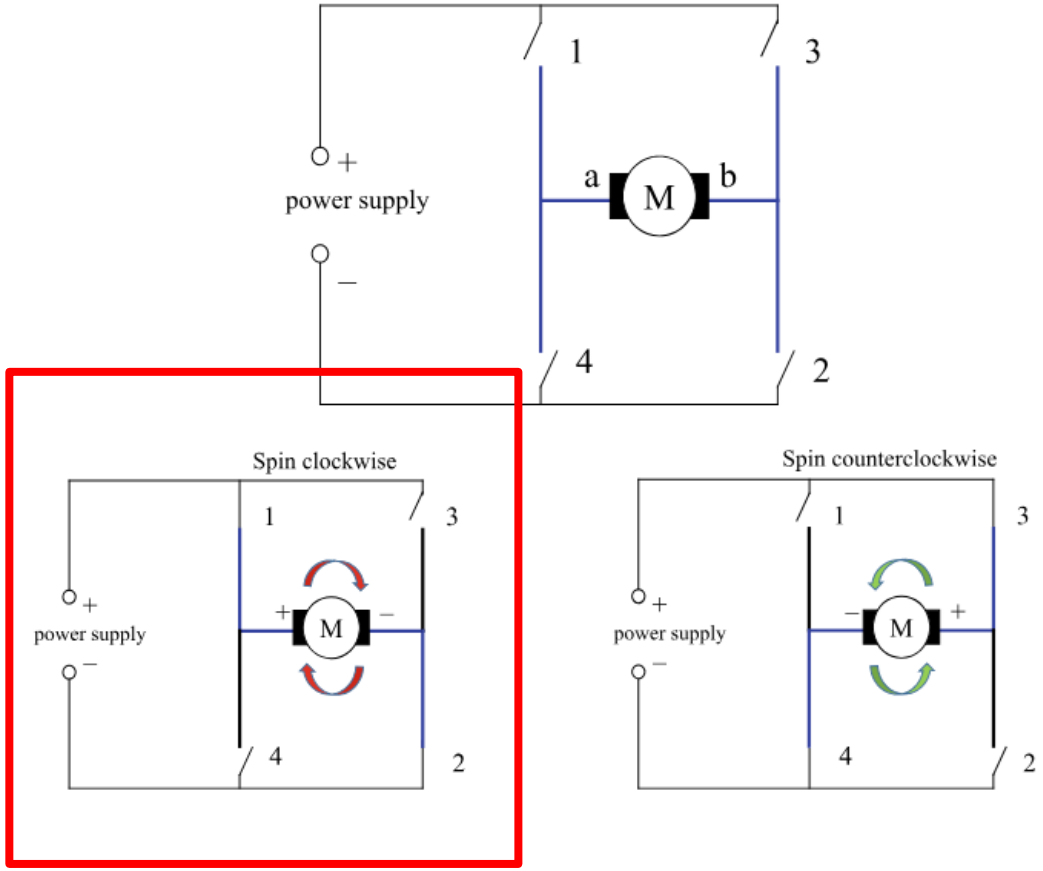


Illustration: [eB1] From Mobile Robots to Autonomous Vehicles with Raspberry Pi and Arduino

H-Bridge

"An electronic circuit that allows a voltage to be applied across a load, such as a DC motor, in either direction."

Controlling a H-Bridge

- **Coasting** (Motor Freewheeling)**:**
  - All switches open (1, 2, 3, and 4)
  - No current flows through the motor.

- **Forward Direction** (Clockwise Rotation)
  - **Switches Closed:** 1 and 2, **Switches Open:** 3 and 4
  - Current flows from the power supply, through 1, across the motor (from one terminal to the other), and then through 2 to ground.

- **Reverse Direction** (Counterclockwise Rotation)
  - **Switches Closed:** 3 and 4, **Switches Open:** 1 and 2
  - Current flows from the power supply, through 3, across the motor in the opposite direction, and then through 4 to ground.

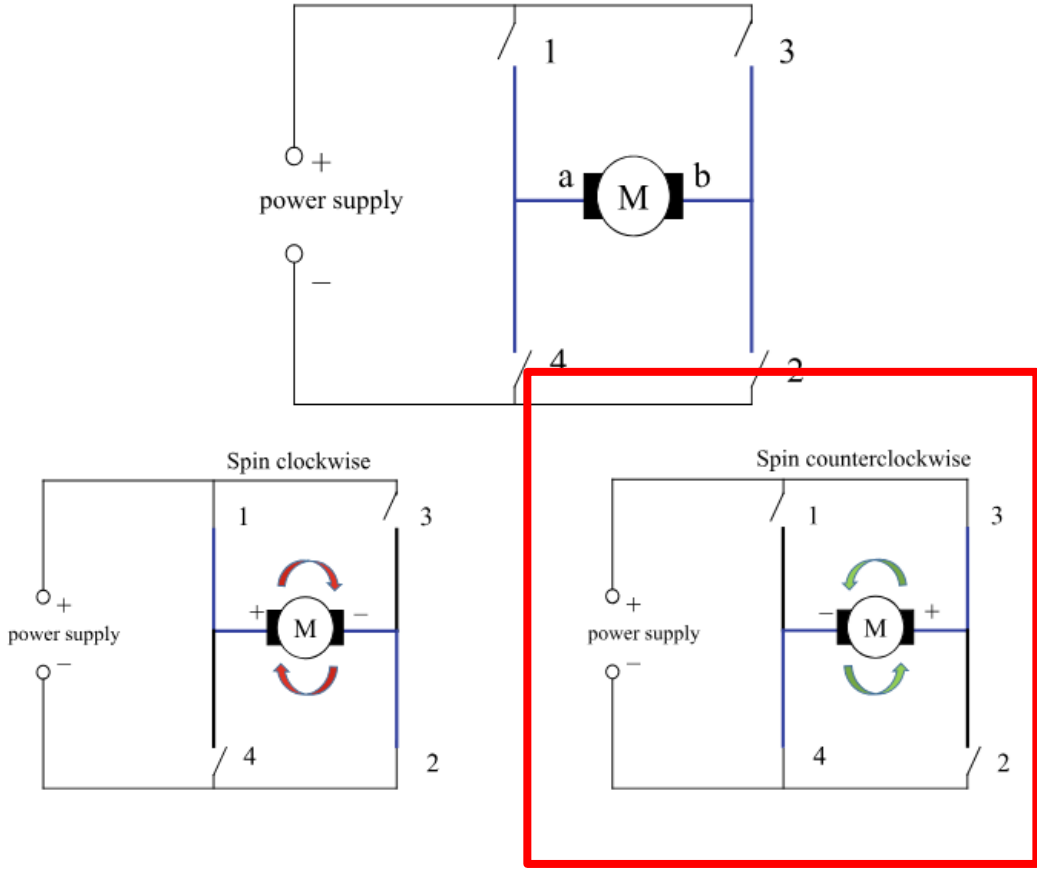
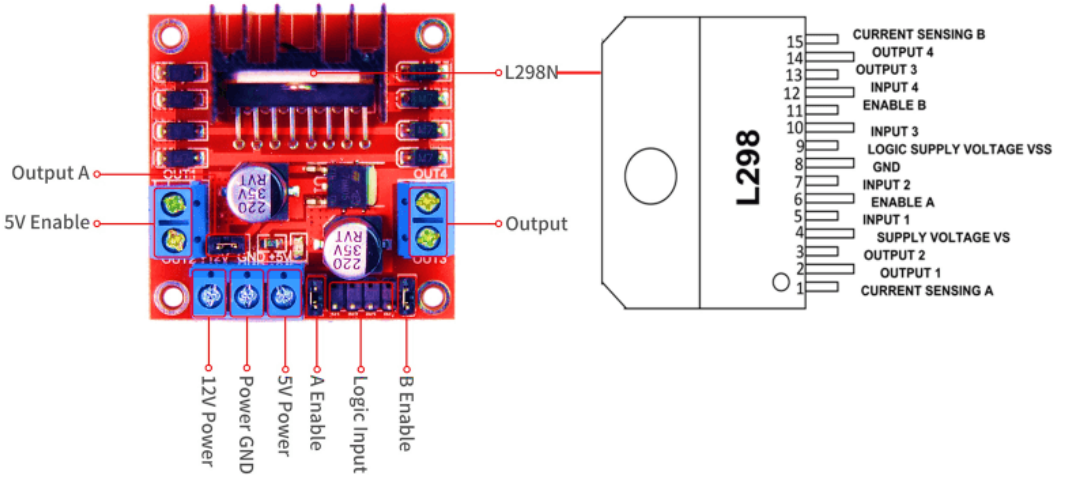


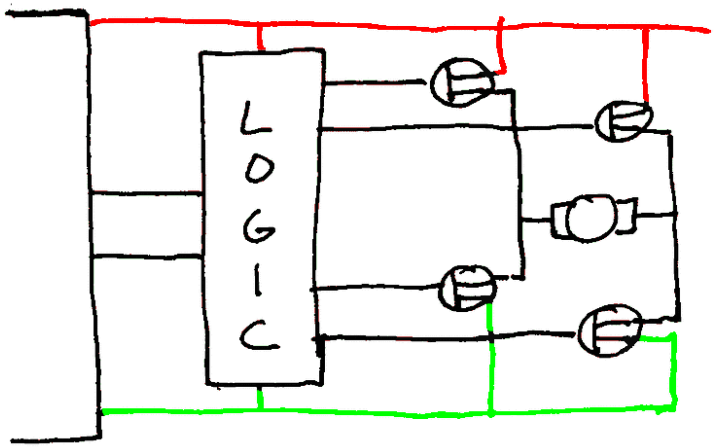
Illustration: [eB1] From Mobile Robots to Autonomous Vehicles with Raspberry Pi and Arduino

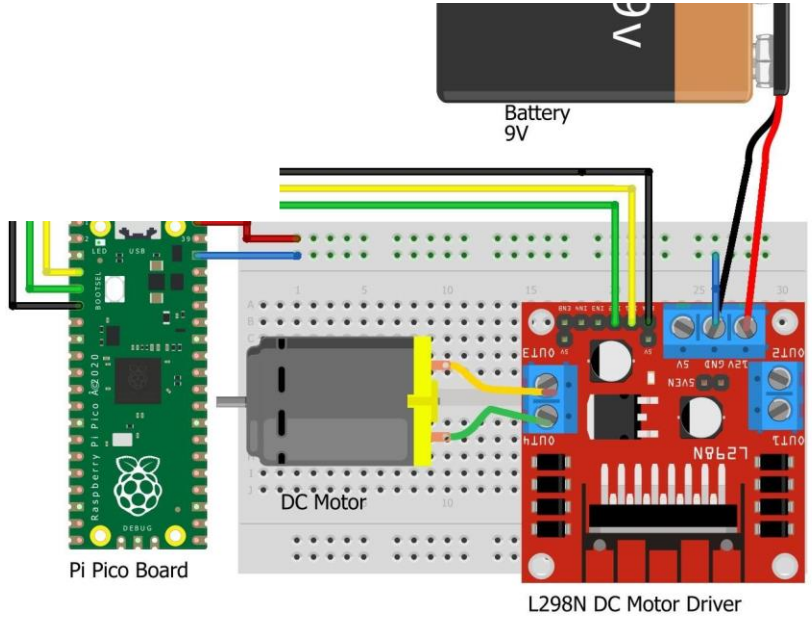
H-Bridge

"An electronic circuit that allows a voltage to be applied across a load, such as a DC motor, in either direction."

- **Example:** L298N DC Driver (use `Pin` class as interface)





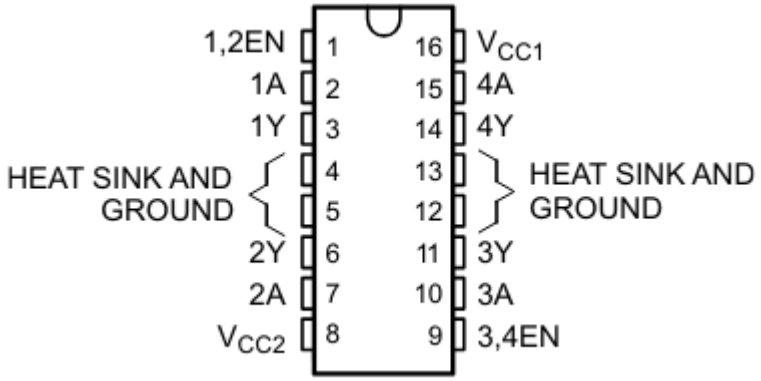


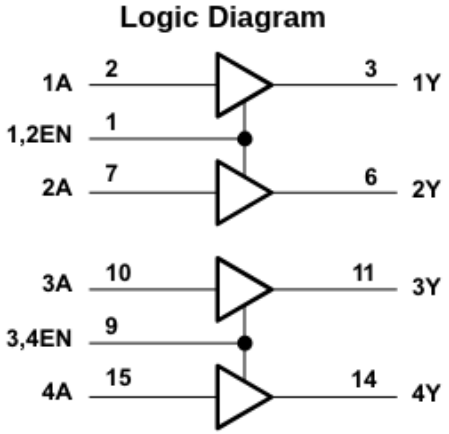
Illustrations: from Google Images

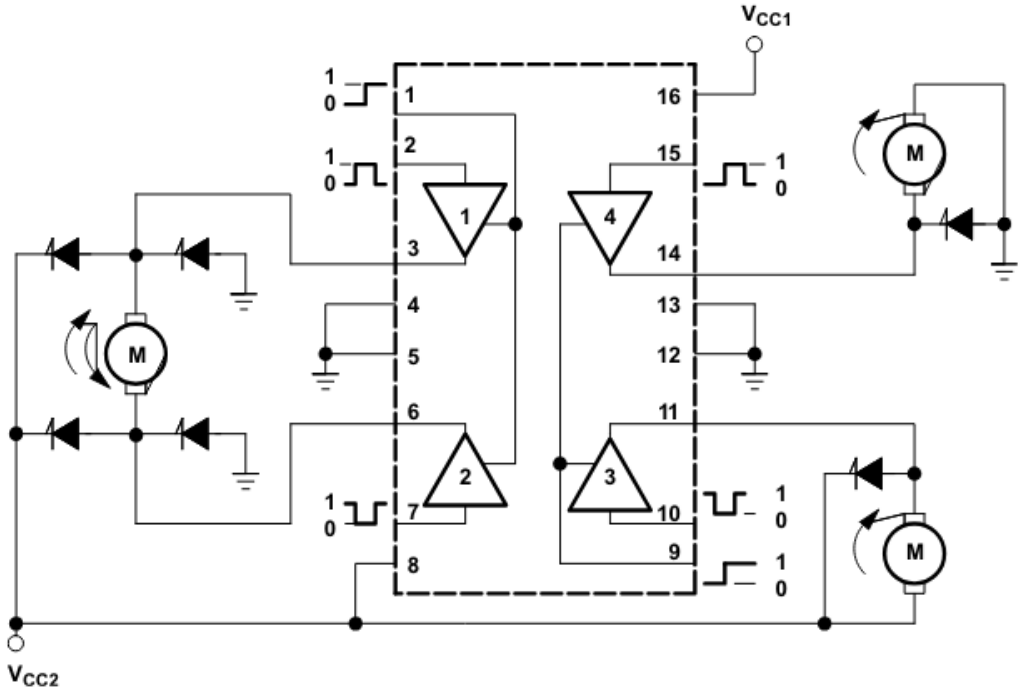
H-Bridge

“An electronic circuit that allows a voltage to be applied across a load, such as a DC motor, in either direction.”

- **Example:** L293x Quadruple Half-H Drivers (Test PCB)
  - Datasheet: https://www.ti.com/lit/ds/symlink/l293.pdf







Pulse Width Modulation (PWM)

Pulse Width Modulation (PWM)

"A digital technique to control a signal by repeatedly toggling a signal between a HIGH and a LOW state in a consistent pattern"

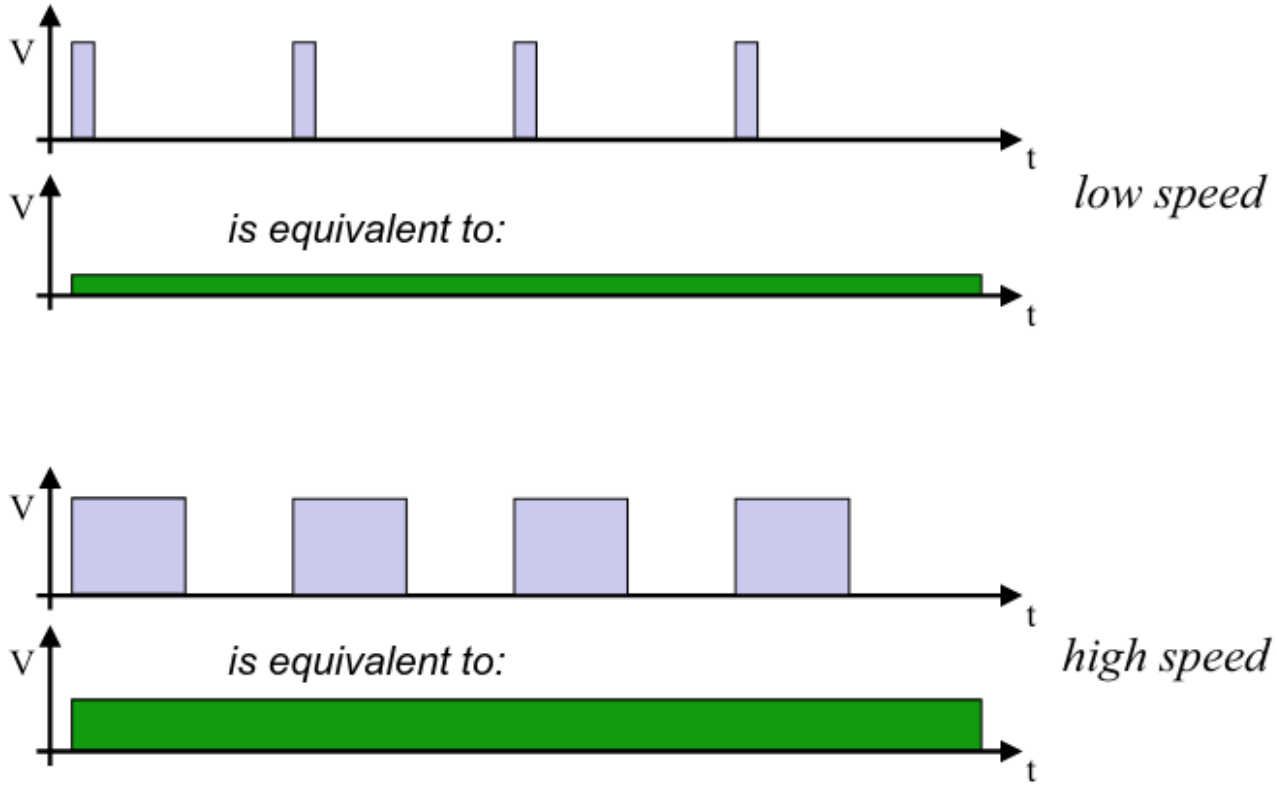


Image: https://www.phidgets.com/education/learn/getting-started-kit-tutorial/dutycycle/

Pulse Width Modulation (PWM)

"A digital technique to control a signal by repeatedly toggling a signal between a HIGH and a LOW state in a consistent pattern"

- **Duty Cycle:** The amount of time the signal stays high (on) compared to the time it stays low (off) within a single cycle.
  - **0% Duty Cycle:** The signal is always off (0% of the time on). The load receives no power.
  - **50% Duty Cycle:** The signal is on for half of the time and off for the other half. The load receives 50% of the maximum possible power.
  - **100% Duty Cycle:** The signal is always on (100% of the time on). The load receives full power.
- **Frequency:** The pulses are generated at a fixed frequency, for example 20 kHz, so they are beyond the human hearing range.

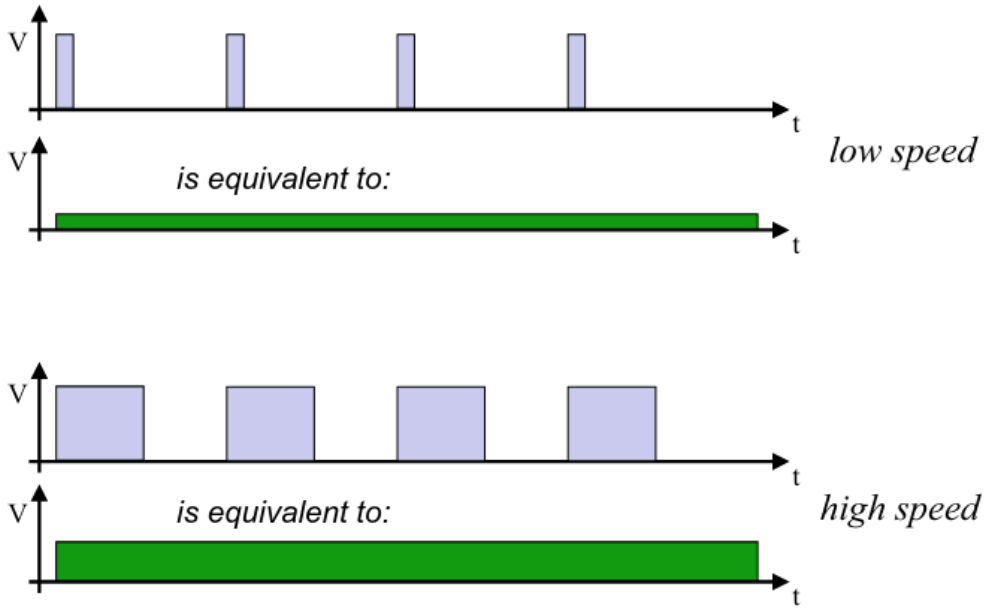


Image: https://www.phidgets.com/education/learn/getting-started-kit-tutorial/dutycycle/

Pulse Width Modulation (PWM)

"A digital technique to control a signal by repeatedly toggling a signal between a HIGH and a LOW state in a consistent pattern"

- **Applications**
  - **LED Dimming:** By adjusting the duty cycle, you can control the brightness of the LED. A lower duty cycle makes the LED dimmer, while a higher duty cycle makes it brighter.
  - **DC Motor Speed Control:** Same principle as the LED; A lower duty cycle makes the motor slower, while a higher duty cycle makes faster. Zero PWM is full stop.

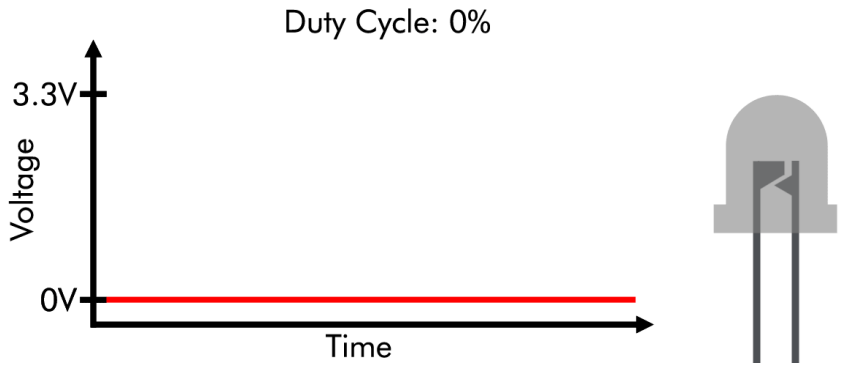
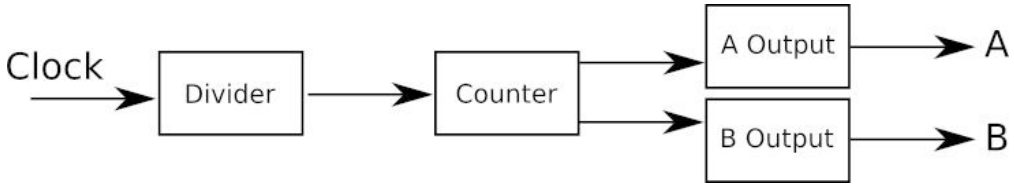


Image: https://www.phidgets.com/education/learn/getting-started-kit-tutorial/dutycycle/

On-chip PWM (RP2040)

- The RP2040 has **8 PWM slices**.
- Each slice can control **two independent channels**, labeled A and B.

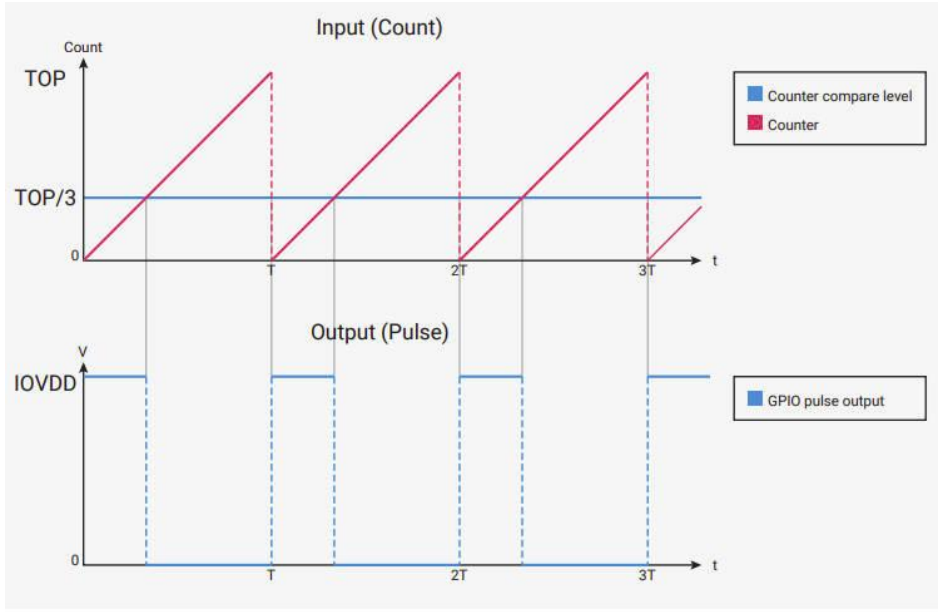


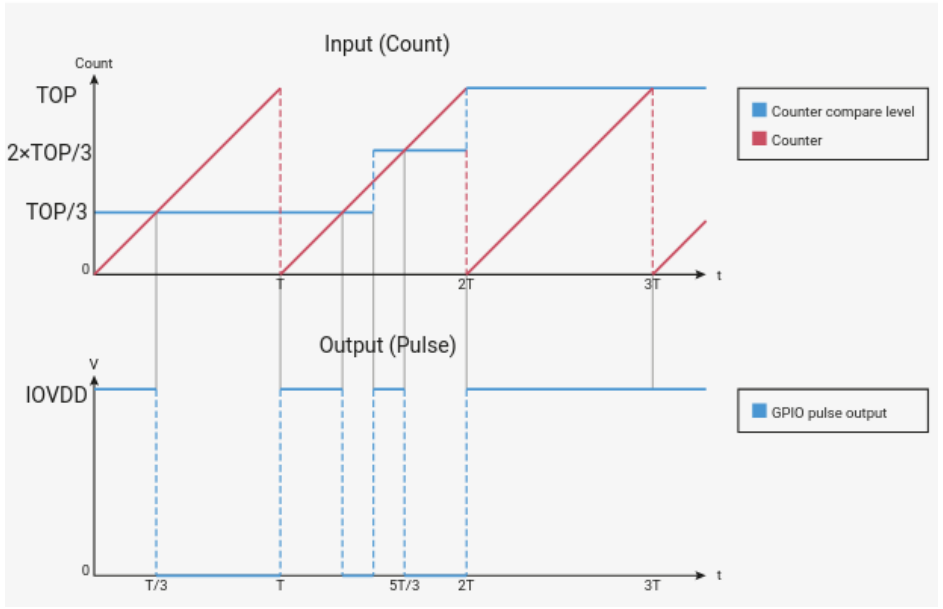
- This allows up to **16 PWM channels** in total, where each channel can be connected to a GPIO pin (subject to pin availability and alternate functions).

- The PWM counters are 16-bit, meaning the duty cycle can be set from 0 to 65535 (CC / Counter compare)

- Default behaviour of a PWM slice
  - Count upward until the maximum value (TOP / 65535 cycles) is reached,
    - if CC is higher than the counter, set output high, else low

GPIO	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
PWM Channel	0A	0B	1A	1B	2A	2B	3A	3B	4A	4B	5A	5B	6A	6B	7A	7B
GPIO	16	17	18	19	20	21	22	23	24	25	26	27	28	29		
PWM Channel	0A	0B	1A	1B	2A	2B	3A	3B	4A	4B	5A	5B	6A	6B		





Source: https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf

Key modules, classes and functions

- PWM **class:** A PWM object provides pulse width modulation output.

- **Constructor**
  - `class machine.PWM(dest, *, freq, duty_u16, duty_ns, invert)`
- **Methods**:
  - `init()`: Initialize a PWM signal on a specific pin.
    - `PWM.init(*, freq, duty_u16, duty_ns)`
    - (You first need to initialize a Pin object, and then pass it to the PWM constructor)
  - `freq()`: Set the frequency of the PWM signal.
  - `duty_u16()`: Set the proportion of time the signal stays high within each period.
    - Accepts a 16-bit integer (0 to 65535):
      - 0 represents a 0% duty cycle
      - 65535 represents a 100% duty cycle
  - `duty_ns()`: Set the current pulse width of the PWM output, as a value in nanoseconds.
  - `deinit()`: De-initializes the PWM object. Disable the PWM output and free up resources.

Classes (PWM module)

- `class Pin` – control I/O pins
- `class Signal` – control and sense external I/O devices
- `class ADC` – analog to digital conversion
- `class ADCBlock` – control ADC peripherals
- `class PWM` – pulse width modulation
- `class UART` – duplex serial communication bus
- `class SPI` – a Serial Peripheral Interface bus protocol (controller side)
- `class I2C` – a two-wire serial protocol
- `class I2S` – Inter-IC Sound bus protocol
- …

`Constants: None`

Example: Import PWM module

```
from machine import Pin, PWM
import time

led = PWM(Pin("GP15")) # Initialize the PWM signal on pin 15
led.freq(1000)

# Gradually increase brightness
for duty in range(0, 65536, 256):
    led.duty_u16(duty)
    time.sleep(0.01)

led.duty_u16(0) # Turn it of once done
```

Key modules, classes and functions

- PWM **class:** A PWM object provides pulse width modulation output.

- **Constructor**
  - `class machine.PWM(dest, *, freq, duty_u16, duty_ns, invert)`
- **Methods**:
  - `init()`: Initialize a PWM signal on a specific pin.
    - `PWM.init(*, freq, duty_u16, duty_ns)`
    - (You first need to initialize a Pin object, and then pass it to the PWM constructor)
  - `freq()`: Set the frequency of the PWM signal.
  - `duty_u16()`: Set the proportion of time the signal stays high within each period.
    - Accepts a 16-bit integer (0 to 65535):
      - 0 represents a 0% duty cycle
      - 65535 represents a 100% duty cycle
  - `duty_ns()`: Set the current pulse width of the PWM output, as a value in nanoseconds.
  - `deinit()`: De-initializes the PWM object. Disable the PWM output and free up resources.

Classes (PWM module)

- `class Pin` – control I/O pins
- `class Signal` – control and sense external I/O devices
- `class ADC` – analog to digital conversion
- `class ADCBlock` – control ADC peripherals
- `class PWM` – pulse width modulation
- `class UART` – duplex serial communication bus
- `class SPI` – a Serial Peripheral Interface bus protocol (controller side)
- `class I2C` – a two-wire serial protocol
- `class I2S` – Inter-IC Sound bus protocol
- …

`Constants: None`

Example: Import PWM module

```
from machine import Pin, PWM
import time

led = PWM(Pin("GP15")) # Initialize the PWM signal on pin 15
led.freq(1000)

# Gradually increase brightness
for duty in range(0, 65536, 256):
    led.duty_u16(duty)
    time.sleep(0.01)

led.duty_u16(0) # Turn it of once done
```

Demo using an LED and EMG49

Servo Motors

Servo motor

“A specialized type of motor that is designed for precise control of angular or linear position, velocity, and acceleration”

(Typically: cannot perform a continuous rotation like a normal DC motor)

- **Range:** ±90° from its middle position (depending on the model)

Image: https://www.phidgets.com/education/learn/getting-started-kit-tutorial/dutycycle/

Servo motor

“A specialized type of motor that is designed for precise control of angular or linear position, velocity, and acceleration”

(Typically: cannot perform a continuous rotation like a normal DC motor)

- **Range:** ±90° from its middle position (depending on the model)
- **Basic Components of a Servo Motor**
  - **Motor**: typically a DC motor
  - **Control Circuit:** interprets input signals (usually PWM) and adjusts the motor's operation to reach the target position.
  - **Position Sensor (feedback):** a potentiometer or encoder, is used to measure the current position of the motor shaft.
  - **(optional) Gearbox:** to reduce the motor’s speed and increase torque.

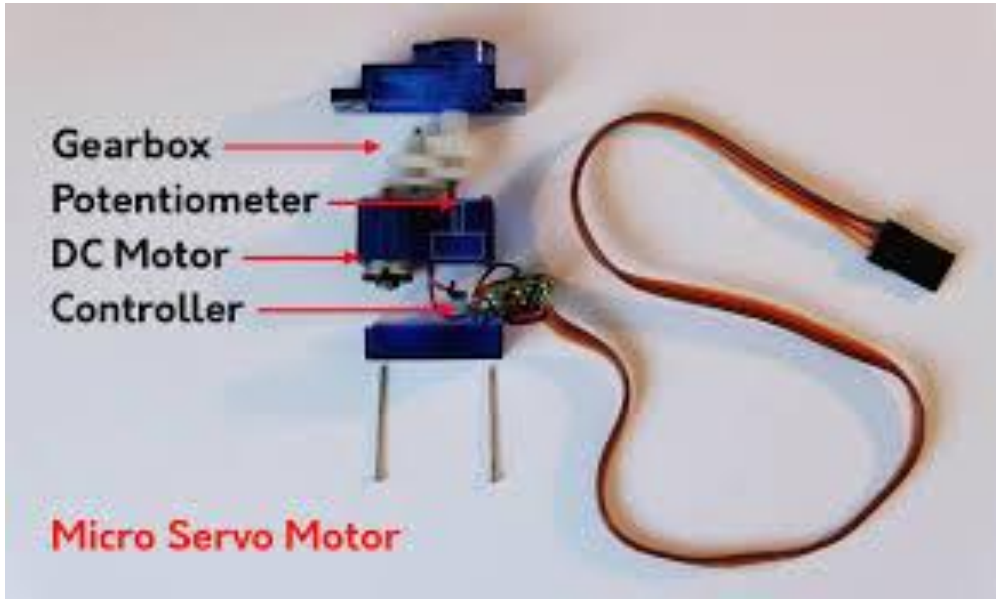


Image: https://www.phidgets.com/education/learn/getting-started-kit-tutorial/dutycycle/

Servo motor

"A specialized type of motor that is designed for precise control of angular or linear position, velocity, and acceleration"

- **Control of a Servo Motor** (using PWM)
  - **Typically three wires:**
    - Vcc (power input),
    - Gnd (ground), and
    - Control signal (PWM)
      - …but not any PWM signal…

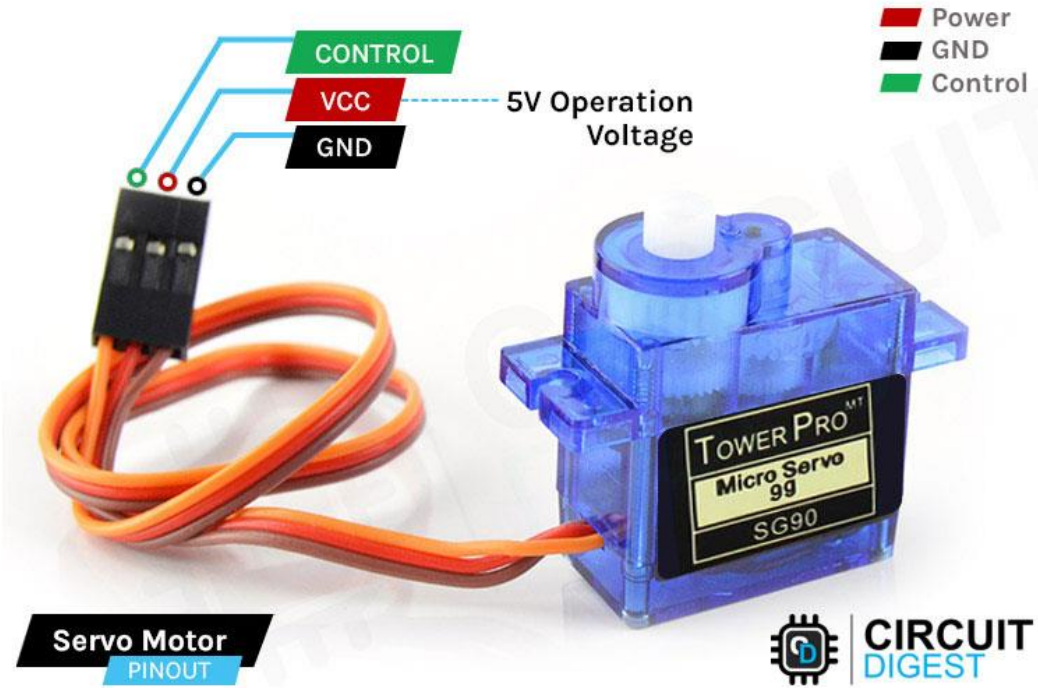


Image: https://circuitdigest.com/microcontroller-projects/interface-servo-motor-with-arduino

Servo motor

"A specialized type of motor that is designed for precise control of angular or linear position, velocity, and acceleration"

- **Control signal**
  - **Frequency:** The PWM signal used for servos always has a **frequency of 50 Hz** (pulses are generated every 20 ms).
  - **Pulse Width:** The width of the pulse within each PWM cycle determines the target position of the servo.

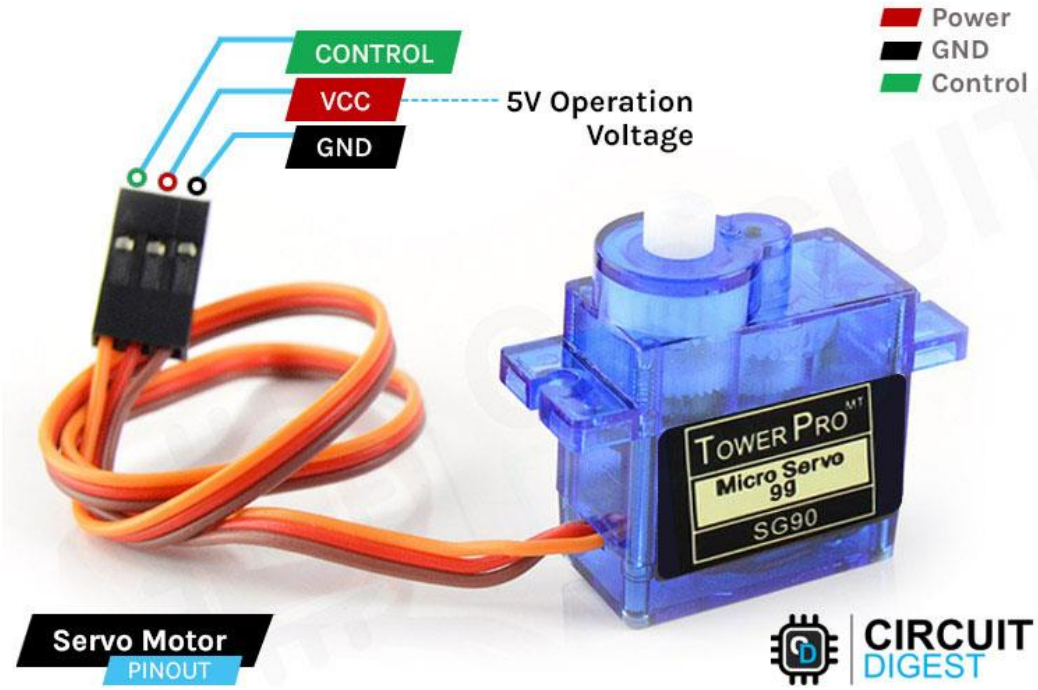
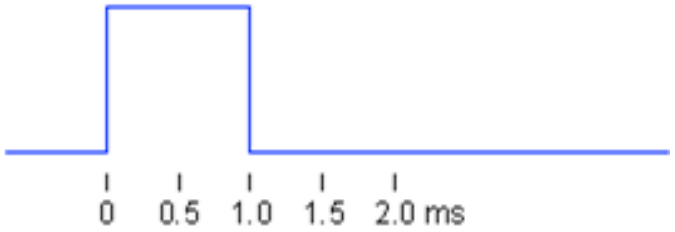


Image: https://circuitdigest.com/microcontroller-projects/interface-servo-motor-with-arduino

Servo motor

"A specialized type of motor that is designed for precise control of angular or linear position, velocity, and acceleration"

- **Pulse Width:** Typically, the pulse width ranges from 1 ms to 2 ms within a **20 ms** cycle.
  - **1 ms pulse:** Positions the motor shaft at one extreme (e.g., 0 degrees).
  - **1.5 ms pulse:** Positions the motor shaft at the midpoint (e.g., 90 degrees).
  - **2 ms pulse:** Positions the motor shaft at the other extreme (e.g., 180 degrees).



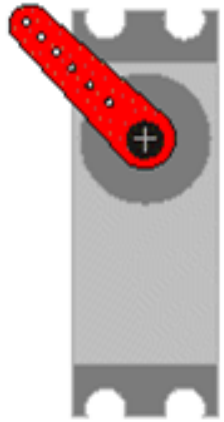


Image: https://kroboblogs.wordpress.com/2016/11/17/servo-motor/

Key modules, classes and functions

- PWM **class:** A PWM object provides pulse width modulation output.

- **Constructor**
  - `class machine.PWM(dest, *, freq, duty_u16, duty_ns, invert)`
- **Methods**:
  - `init()`: Initialize a PWM signal on a specific pin.
    - `PWM.init(*, freq, duty_u16, duty_ns)`
    - (You first need to initialize a Pin object, and then pass it to the PWM constructor)
  - `freq()`: Set the frequency of the PWM signal.
  - `duty_u16()`: Set the proportion of time the signal stays high within each period.
    - Accepts a 16-bit integer (0 to 65535):
      - 0 represents a 0% duty cycle
      - 65535 represents a 100% duty cycle
  - `duty_ns()`: Set the current pulse width of the PWM output, as a value in nanoseconds.
  - `deinit()`: De-initializes the PWM object. Disable the PWM output and free up resources.

Example: Import PWM module

```
from machine import Pin, PWM
import time

# Initialize the servo on GP15 and set PWM frequency to 50Hz
servo_pin = Pin("GP15")
servo = PWM(servo_pin)
servo.freq(50)

# Calculate min and max duty cycles for 0° and 180° positions
duty = 65535
max_duty = int(2.0 / 20 * duty)
min_duty = int(1.0 / 20 * duty)

# Move servo to 0°, wait, then move to 180°
servo.duty_u16(min_duty)
time.sleep(5)
servo.duty_u16(max_duty)
```

Key modules, classes and functions

- PWM **class:** A PWM object provides pulse width modulation output.
- **Constructor**
  - `class machine.PWM(dest, *, freq, duty_u16, duty_ns, invert)`
- **In this example**:
  - `duty = 65535` represents the full range of the 16-bit PWM (from 0 to 65535).
  - `max_duty` is calculated for the 2.0 ms pulse width, corresponding to 180°.
  - `min_duty` is calculated for the 1.0 ms pulse width, corresponding to 0°.
  - Moving the servo:
    - `servo.duty_u16(min_duty)` sets the duty cycle to `min_duty`, moving the servo to its 0° position,
    - then pauses for 5 seconds, and then
    - `servo.duty_u16(max_duty)` sets the duty cycle to `min_duty`, moving the servo to its 180° position,

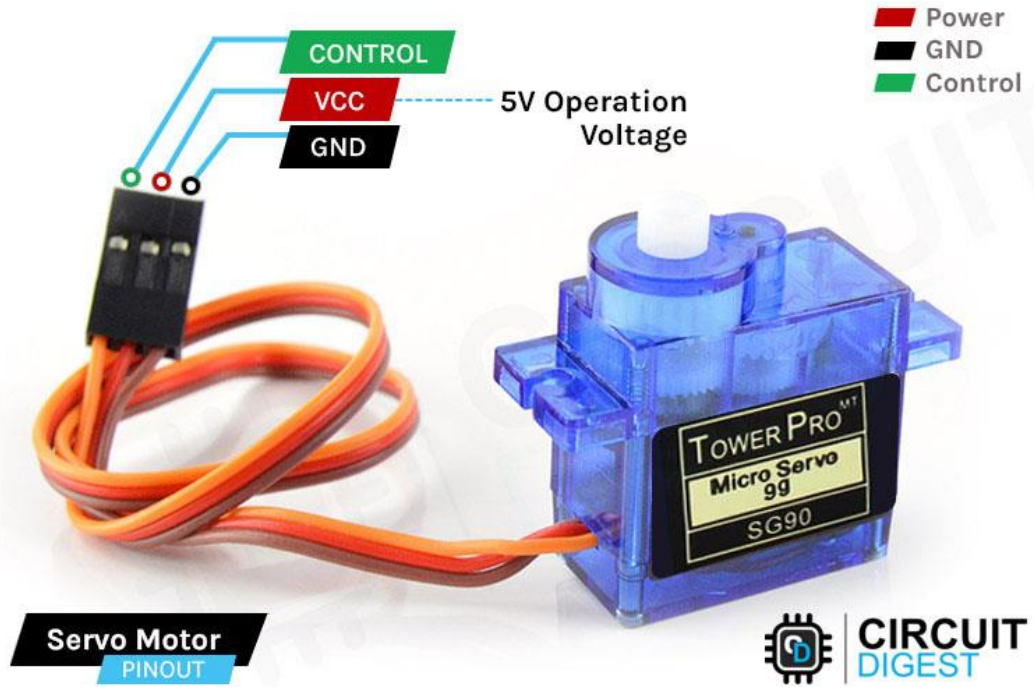
Example: Import PWM module

```
from machine import Pin, PWM
import time

# Initialize the servo on GP15 and set PWM frequency to 50Hz
servo_pin = Pin("GP15")
servo = PWM(servo_pin)
servo.freq(50)

# Calculate min and max duty cycles for 0° and 180° positions
duty = 65535
max_duty = int(2.0 / 20 * duty)
min_duty = int(1.0 / 20 * duty)

# Move servo to 0°, wait, then move to 180°
servo.duty_u16(min_duty)
time.sleep(5)
servo.duty_u16(max_duty)
```

Key modules, classes and functions

- PWM **class:** A PWM object provides pulse width modulation output.

- **Actual (full) range for the SG90:** from 0.7 ms → 2.4 ms



Example: Import PWM module

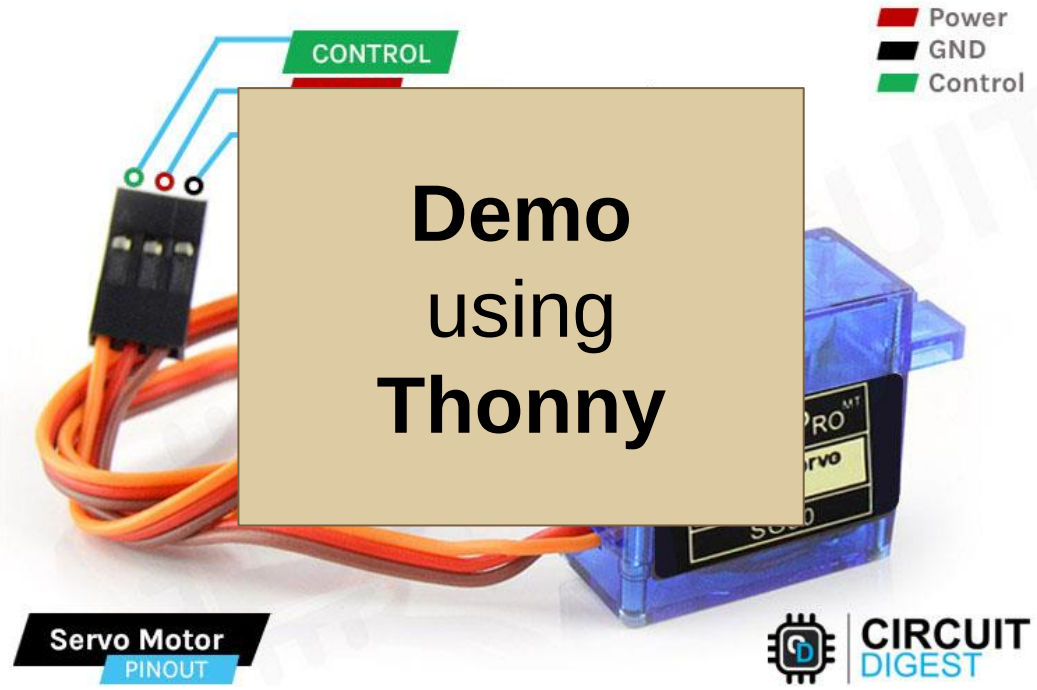
```
from machine import Pin, PWM
import time

# Initialize the servo on GP15 and set PWM frequency to 50Hz
servo_pin = Pin("GP15")
servo = PWM(servo_pin)
servo.freq(50)

# Calculate min and max duty cycles for 0° and 180° positions
duty = 65535
max_duty = int(2.4 / 20 * duty)
min_duty = int(0.7 / 20 * duty)

# Move servo to 0°, wait, then move to 180°, wait, and back
servo.duty_u16(min_duty)
time.sleep(5)
servo.duty_u16(max_duty)
time.sleep(5)
servo.duty_u16(min_duty)
```

Image: https://circuitdigest.com/microcontroller-projects/interface-servo-motor-with-arduino

Key modules, classes and functions

- PWM **class:** A PWM object provides pulse width modulation output.

- **Actual (full) range for the SG90:** from 0.7 ms → 2.4 ms



Example: Import PWM module

```
from machine import Pin, PWM
import time

# Initialize the servo on GP15 and set PWM frequency to 50Hz
servo_pin = Pin("GP15")
servo = PWM(servo_pin)
servo.freq(50)

# Calculate min and max duty cycles for 0° and 180° positions
duty = 65535
max_duty = int(2.4 / 20 * duty)
min_duty = int(0.7 / 20 * duty)

# Move servo to 0°, wait, then move to 180°, wait, and back
servo.duty_u16(min_duty)
time.sleep(5)
servo.duty_u16(max_duty)
time.sleep(5)
servo.duty_u16(min_duty)
```

Image: https://circuitdigest.com/microcontroller-projects/interface-servo-motor-with-arduino

Stepper Motors

Stepper motors

“Move in discrete steps, allowing precise control over position, speed, and acceleration”

- Designed to move in discrete steps rather than continuously rotating like traditional DC motors

- **Basic Components of a Servo Motor**
  - **Rotor:** The rotor is the rotating part of the motor, typically made of a permanent magnet or soft iron.

  - **Stator:** The stator is the stationary part surrounding the rotor, composed of multiple coils (also called windings). These coils are energized in a specific sequence to create a magnetic field that interacts with the rotor, causing it to move.

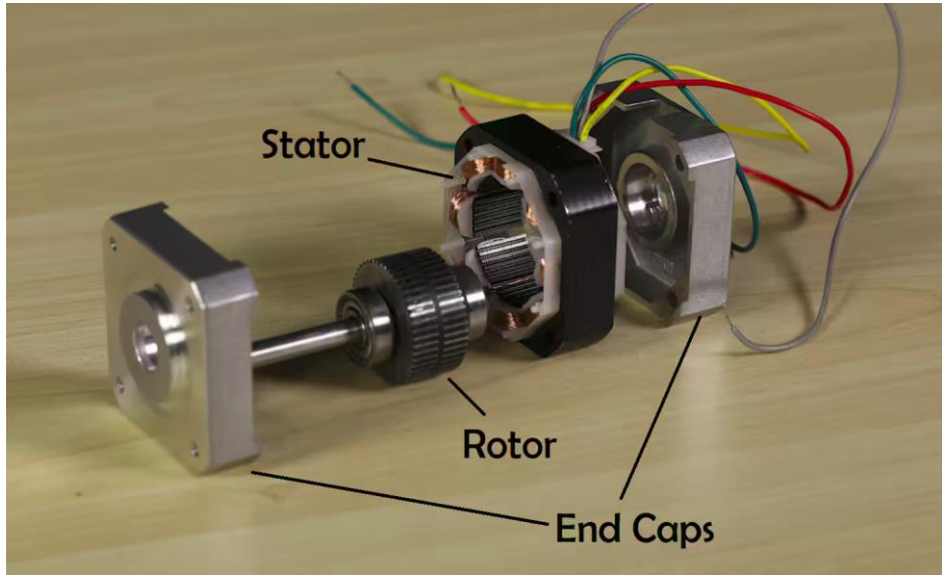
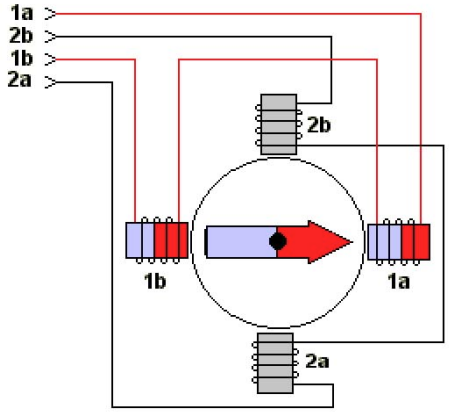


Image: https://www.digikey.dk/da/blog/the-basics-of-stepper-motors

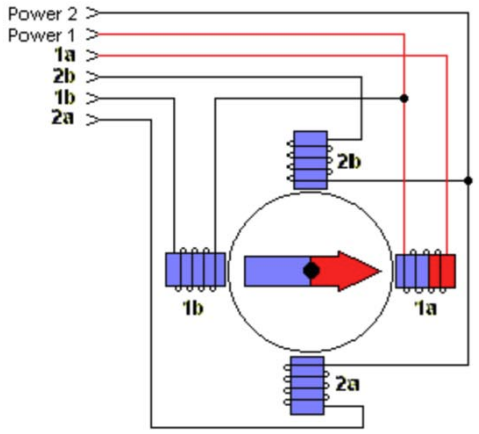
Stepper motors

"Move in discrete steps, allowing precise control over position, speed, and acceleration"

- **Stepper motor types** (typical)
  a. Unipolar Stepper Motors
  b. Bipolar Stepper Motors



Conceptual Model of Bipolar Stepper Motor



Conceptual Model of Unipolar Stepper Motor

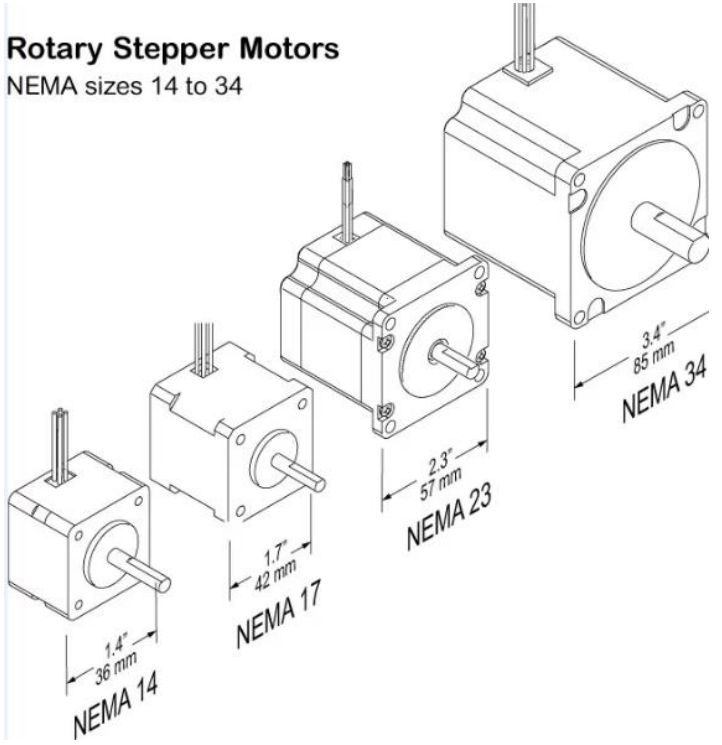


Image: https://fabacademy.org/archives/2014/students/begle.moritz/week13.html and from Google Images

Stepper motors

“Move in discrete steps, allowing precise control over position, speed, and acceleration”

- **Stepping** (key concepts)

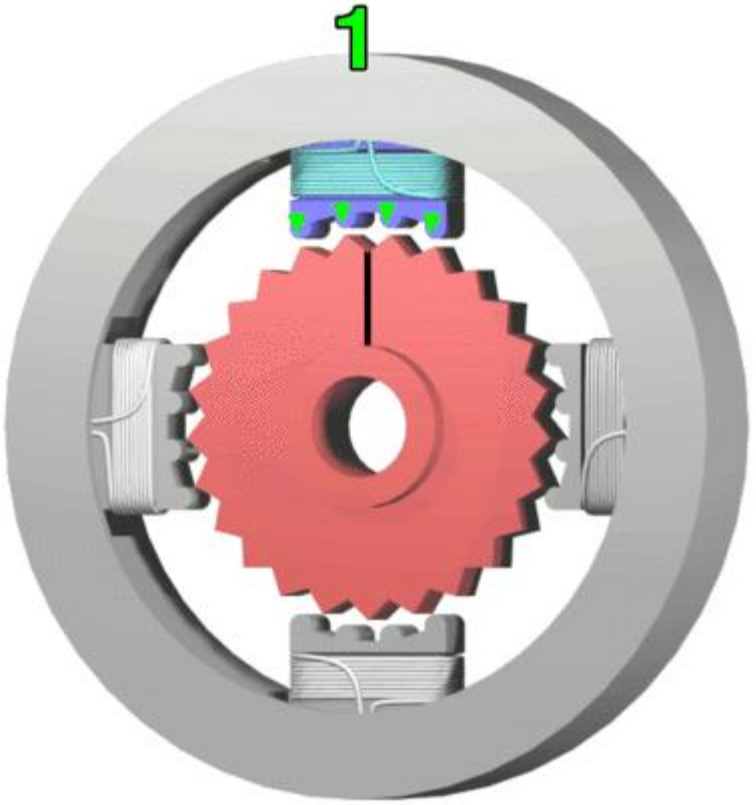


Image: https://en.wikipedia.org/wiki/Stepper_motor

Stepper motors

“Move in discrete steps, allowing precise control over position, speed, and acceleration”

- **Stepping** (key concepts)
  - A stepper motor moves in fixed angular increments called steps.

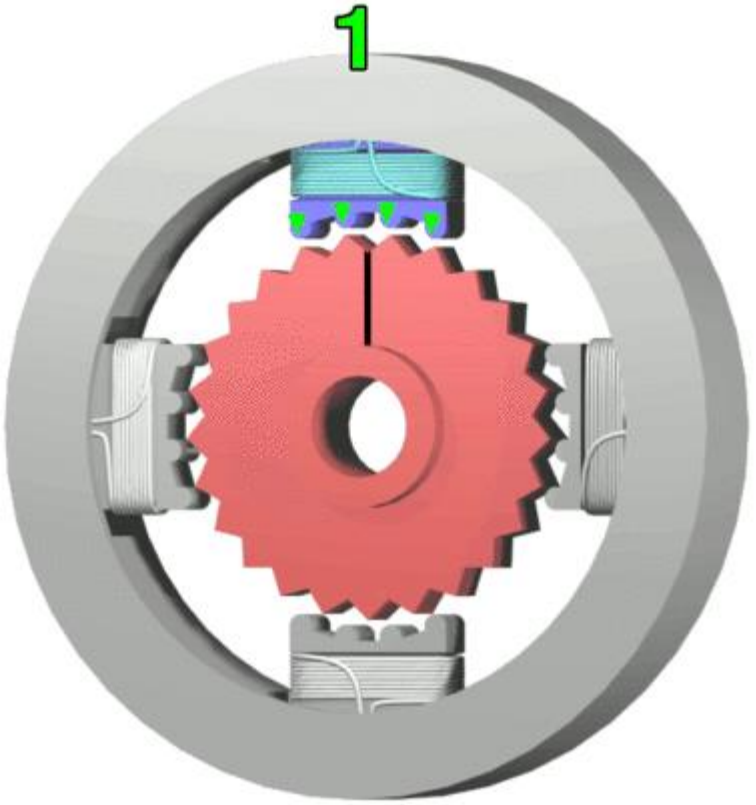


Image: https://en.wikipedia.org/wiki/Stepper_motor

Stepper motors

“Move in discrete steps, allowing precise control over position, speed, and acceleration”

- **Stepping** (key concepts)
  - A stepper motor moves in fixed angular increments called steps.
  - Each step is controlled by **energizing the coils in a specific sequence**, which creates a rotating magnetic field that pulls the rotor into the next position.

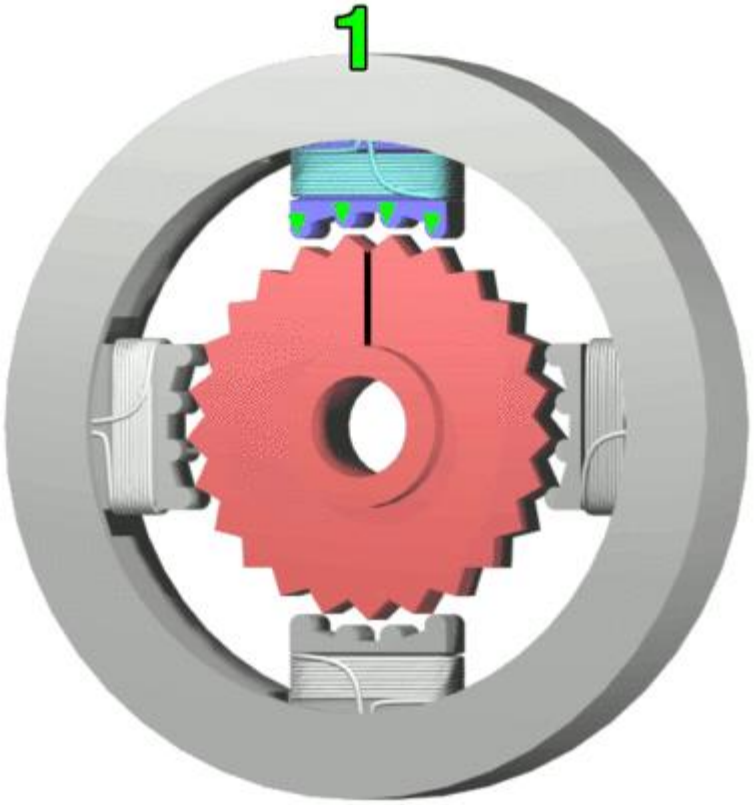


Image: https://en.wikipedia.org/wiki/Stepper_motor

Stepper motors

“Move in discrete steps, allowing precise control over position, speed, and acceleration”

- **Stepping** (key concepts)
  - A stepper motor moves in fixed angular increments called steps.
  - Each step is controlled by **energizing the coils in a specific sequence**, which creates a rotating magnetic field that pulls the rotor into the next position.
  - …by changing the order and timing of the pulses to the coils, you can control the motor's direction, speed, and position.

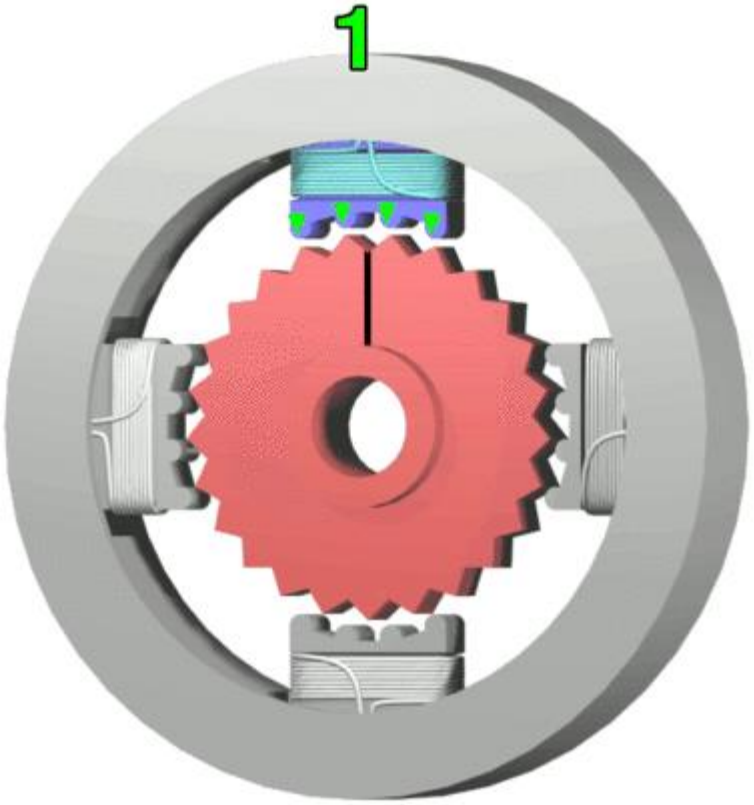
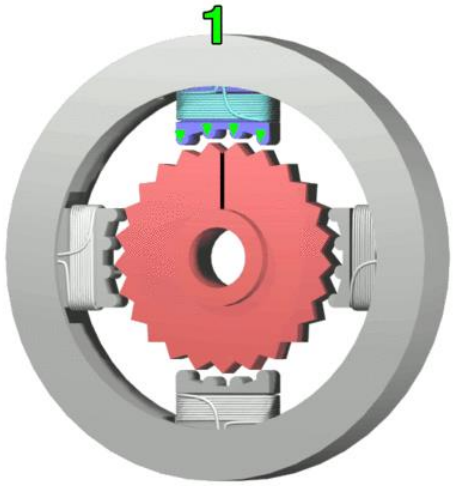


Image: https://en.wikipedia.org/wiki/Stepper_motor

Stepper motors



“Move in discrete steps, allowing precise control over position, speed, and acceleration”

- **Stepping** (key concepts)
  - A stepper motor moves in fixed angular increments called steps.
    - Example: **17HE19-2004S**
      - **Motor Type:** Bipolar Stepper
      - **Step Angle:** 1.8deg (200 steps per revolution)
      - **Number of Leads:** 4

A+	A-	B+	B-
Black	Blue	Green	Red

- **Datasheet:**

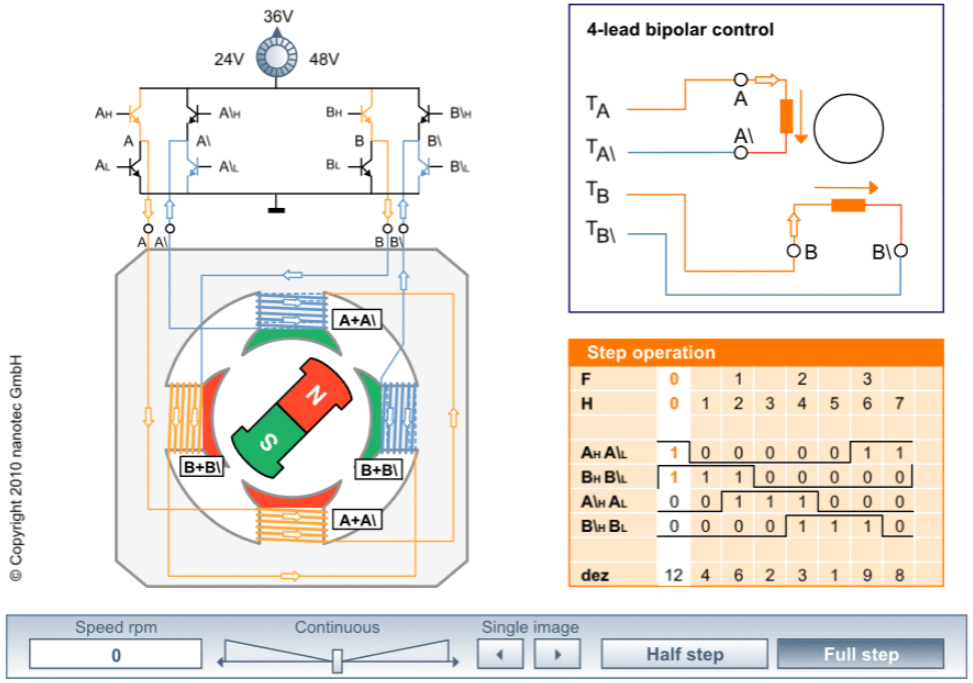
https://www.omc-stepperonline.com/e-series-nema-17-bipolar-55ncm-77-88oz-in-2a-42x48mm-4-wires-w-1m-cable-connector-17he19-2004s

Image: https://en.wikipedia.org/wiki/Stepper_motor

Stepper motors

"Move in discrete steps, allowing precise control over position, speed, and acceleration"

- **Stepping Modes:**
  - **Full Step**: The motor moves one full step per pulse. This is the simplest and most common mode.
  - **Half Step**: The motor moves half a step per pulse, offering more precision but less torque.
  - **Microstepping**: By controlling the current through the coils, the motor can be made to take fractional steps, providing smoother motion and higher resolution.



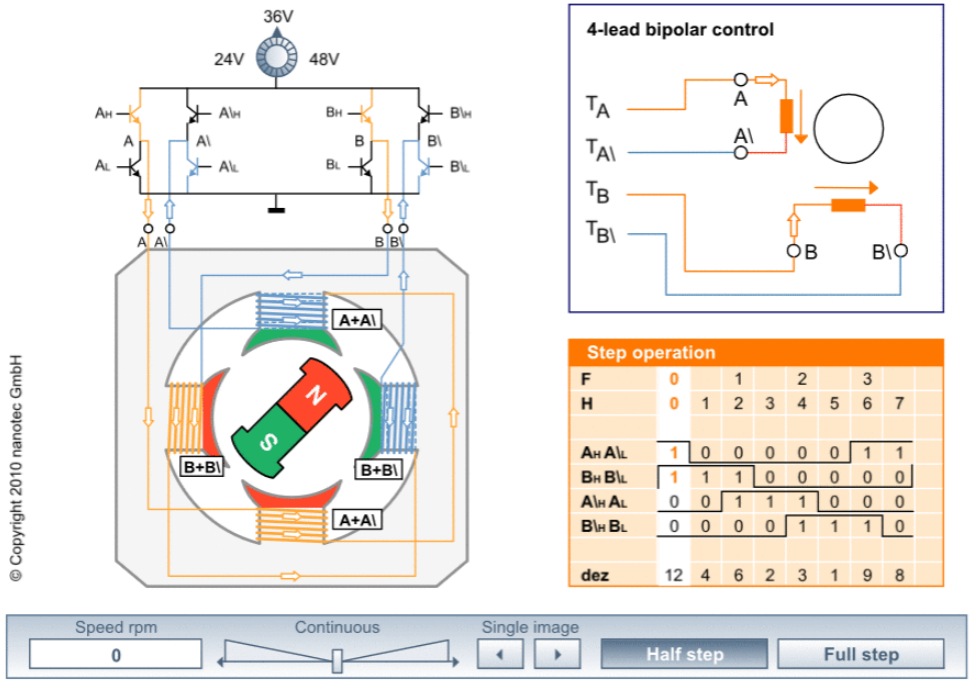
Full step

Image: https://www.nanotec.com/eu/en/knowledge-base/stepper-motor-animation#

Stepper motors

"Move in discrete steps, allowing precise control over position, speed, and acceleration"

- **Stepping Modes:**
  - **Full Step**: The motor moves one full step per pulse. This is the simplest and most common mode.
  - **Half Step**: The motor moves half a step per pulse, offering more precision but less torque.
  - **Microstepping**: By controlling the current through the coils, the motor can be made to take fractional steps, providing smoother motion and higher resolution.



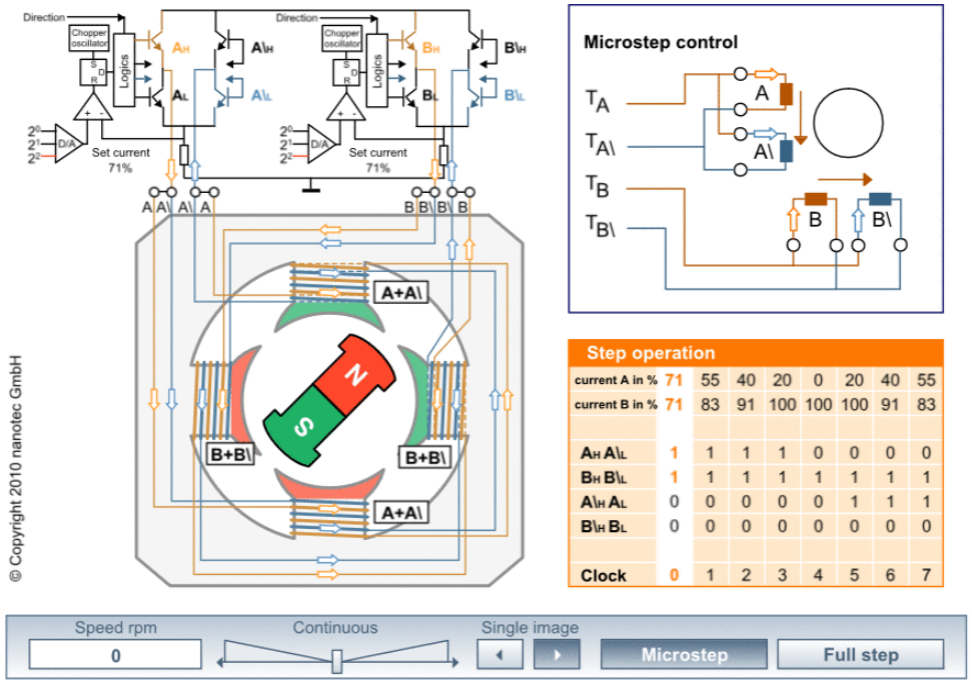
Half step

Image: https://www.nanotec.com/eu/en/knowledge-base/stepper-motor-animation#

Stepper motors

"Move in discrete steps, allowing precise control over position, speed, and acceleration"

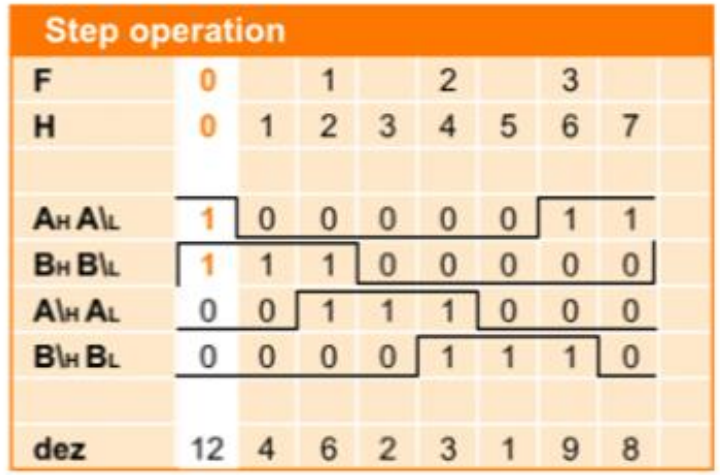
- **Stepping Modes:**
  - **Full Step**: The motor moves one full step per pulse. This is the simplest and most common mode.
  - **Half Step**: The motor moves half a step per pulse, offering more precision but less torque.
  - **Microstepping**: By controlling the current through the coils, the motor can be made to take fractional steps, providing smoother motion and higher resolution.



Microstepping

Image: https://www.nanotec.com/eu/en/knowledge-base/stepper-motor-animation#

Half step



Grey Code

“a binary numeral system where two successive values differ in only one bit.”

- The stepping sequence pattern resembles a **gray code**
  - A form of binary that uses a different method of incrementing from one number to the next.
    - Only a single bit changes at each transition
  - Reduces the likelihood of errors during the transition.

Decimal	Binary	Gray Code		Stepping sequence
0	0000	0000		1100
1	0001	0001		0100
2	0010	0011		0110
3	0011	0010		0010
4	0100	0110		0011
5	0101	0111		0001
6	0110	0101		1001
7	0111	0100		1000
8	1000	1100		
9	1001	1101		
10	1010	1111		
11	1011	1110		
12	1100	1010		
13	1101	1011		
14	1110	1001		
15	1111	1000		

Image: https://www.nanotec.com/eu/en/knowledge-base/stepper-motor-animation#

Key modules, classes and functions

- `Pin` **class:** A pin object is used to control digital I/O pins

- **Constructor**
  - `class machine.Pin(id, mode=-1, pull=-1, *, value=None, drive=0, alt=-1)`

- **Example**:
  - **Define GPIO Pins**: The GPIO pins connected to the motor driver are defined as outputs.
  - **Full-Step Sequence**: The `full_step_sequence` list contains the binary values for each full step. The sequence energizes two coils at a time, which ensures that the motor moves in full steps.
  - **set_step Function**: This function sets the GPIO pins' states according to the current step in the sequence.
  - **rotate_full_step Function**: This function loops through the `full_step_sequence` to rotate the motor. The `steps` parameter specifies the number of steps, and the `delay` parameter specifies the delay between steps, controlling the speed of the motor.
  - **Turning Off the Coils**: After completing the rotation, the `set_step([0, 0, 0, 0])` function call ensures that all coils are turned off to prevent unnecessary power consumption.
  - **Example Usage**: The motor is rotated 10 steps forward and then 10 steps backward, with a 10ms delay between each step.

Example: Stepper motor

```
from machine import Pin
from time import sleep

# Define the GPIO pins connected to the stepper motor driver
coil_A1 = Pin("GP0", Pin.OUT)
coil_A2 = Pin("GP1", Pin.OUT)
coil_B1 = Pin("GP2", Pin.OUT)
coil_B2 = Pin("GP3", Pin.OUT)

# Define the full-step sequence
full_step_sequence = [
    [1, 1, 0, 0],  # Step 0
    [0, 1, 1, 0],  # Step 1
    [0, 0, 1, 1],  # Step 2
    [1, 0, 0, 1],  # Step 3
]

def set_step(step):
    # Set the GPIO outputs according to the step
    coil_A1.value(step[0])
    coil_A2.value(step[1])
    coil_B1.value(step[2])
    coil_B2.value(step[3])

def rotate_full_step(steps, delay):
    # Rotate the motor for a given number of steps
    for _ in range(steps):
        for step in full_step_sequence:
            set_step(step)
            sleep(delay)
    # Turn off all coils after rotation
    set_step([0, 0, 0, 0])

# Example usage:
rotate_full_step(10, 0.01) # Rotate the motor 10 full steps
```

Key modules, classes and functions

- PWM **class:** A PWM object provides pulse width modulation output.

- **Constructor**
  - `class machine.PWM(dest, *, freq, duty_u16, duty_ns, invert)`

- **Example**:
  - **PWM Setup**: Each GPIO pin connected to the stepper motor is configured as a PWM output. The frequency is set to 16 kHz, which is a good starting point.

  - **Full-Step Sequence with PWM**: The `full_step_sequence` now includes PWM duty cycles instead of simple binary values. The duty cycle values are represented as percentage (0 = 0% → 1 = 100%), which correspond to the motor coil's current level.

  - **set_step Function**: This function sets the PWM duty cycle for each coil according to the current step. The duty cycle is multiplied by 65535 to convert it into the 16-bit format expected by `duty_u16()`.

  - **rotate_full_step Function**: This function rotates the motor for a given number of steps, with a specified delay between each step. It runs the full-step sequence in the forward or reverse direction, depending on the sign of `steps`.

  - **Turning Off the Coils**: After completing the rotation, the `set_step([0, 0, 0, 0])` function call ensures that all coils are turned off to prevent unnecessary power consumption.

  - **Example Usage**: The motor is rotated 10 steps forward and then 10 steps backward, with a 10ms delay between each step.

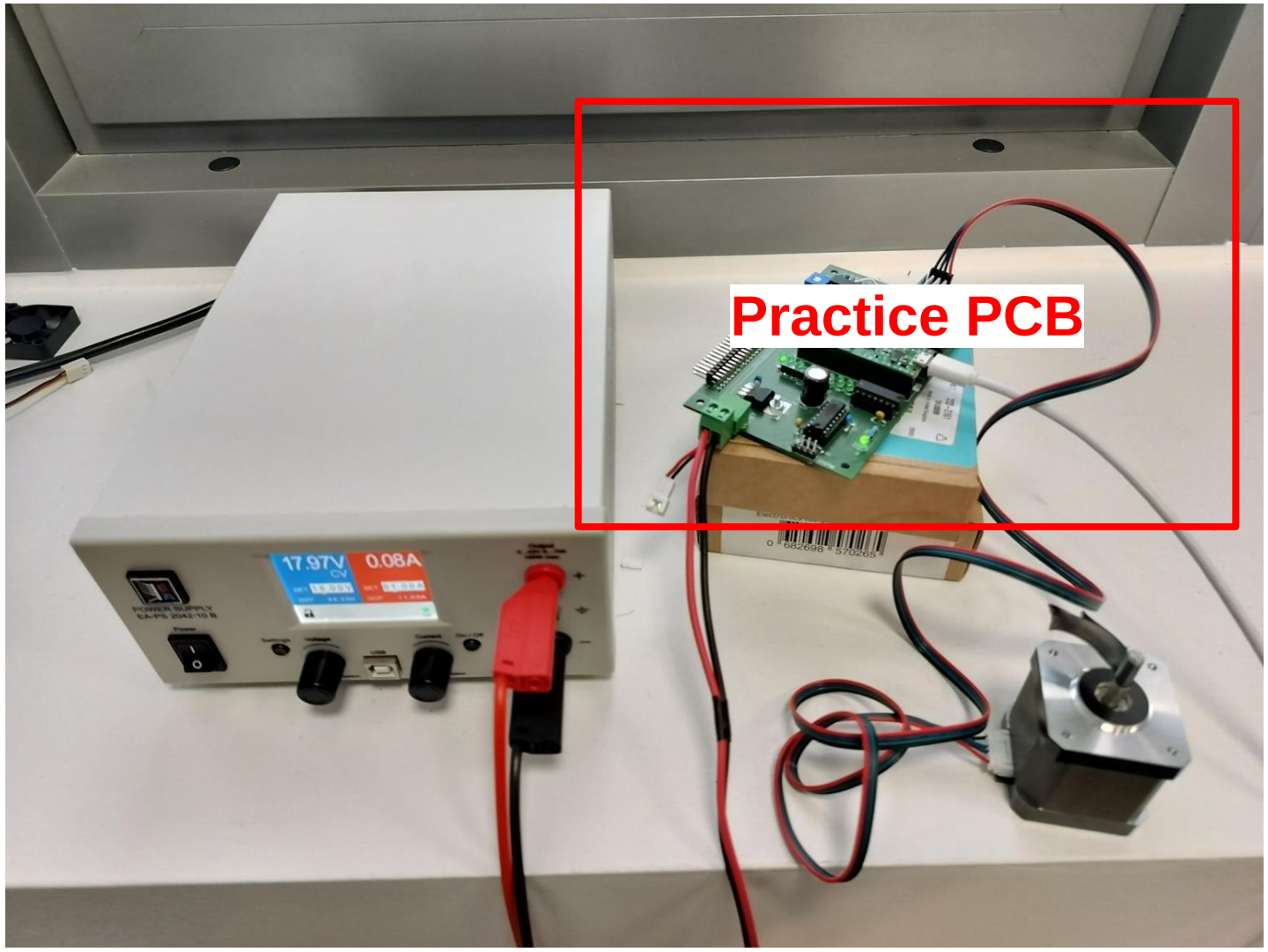
Example: Stepper motor

```
from machine import Pin, PWM
from time import sleep

# Define the GPIO pins connected to the stepper motor driver as PWM
coil_A1 = PWM(Pin("GP0", Pin.OUT))
coil_A2 = PWM(Pin("GP1", Pin.OUT))
coil_B1 = PWM(Pin("GP2", Pin.OUT))
coil_B2 = PWM(Pin("GP3", Pin.OUT))

# Set the PWM frequency (a good starting point is 16 kHz)
coil_A1.freq(16_000)
coil_A2.freq(16_000)
coil_B1.freq(16_000)
coil_B2.freq(16_000)

# Define the full-step sequence with 50% PWM duty cycles
full_step_sequence = [
  [0.5, 0.5, 0, 0],  # Step 0
  [0, 0.5, 0.5, 0],  # Step 1
  [0, 0, 0.5, 0.5],  # Step 2
  [0.5, 0, 0, 0.5],  # Step 3
]

def set_step(step):  # Set the PWM duty cycle according to the step
   coil_A1.duty_u16(int(step[0] * 65535))  # Convert to 16-bit duty cycle
   coil_A2.duty_u16(int(step[1] * 65535))
   coil_B1.duty_u16(int(step[2] * 65535))
   coil_B2.duty_u16(int(step[3] * 65535))

def rotate_full_step(steps, delay): # Rotate the motor
   for _ in range(abs(steps)):
       for step in full_step_sequence:
           set_step(step)
           sleep(delay)
   set_step([0, 0, 0, 0]) # Turn off all coils after rotation

rotate_full_step(50, 0.01)  # Rotate the motor 50 full steps seqeunces
```

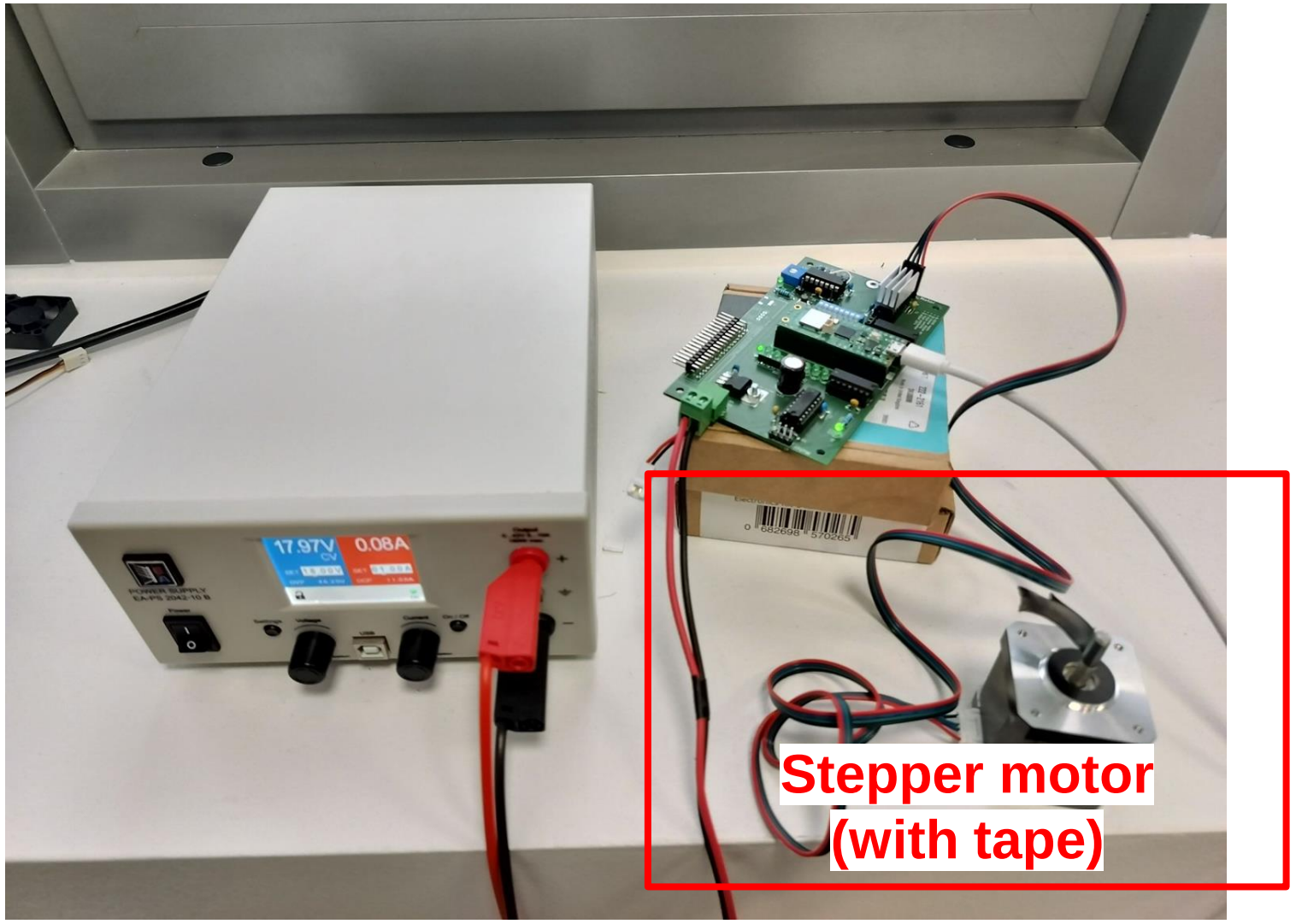
Extra Credit Activities #2:
Stepper Motor Controller class

Test setup

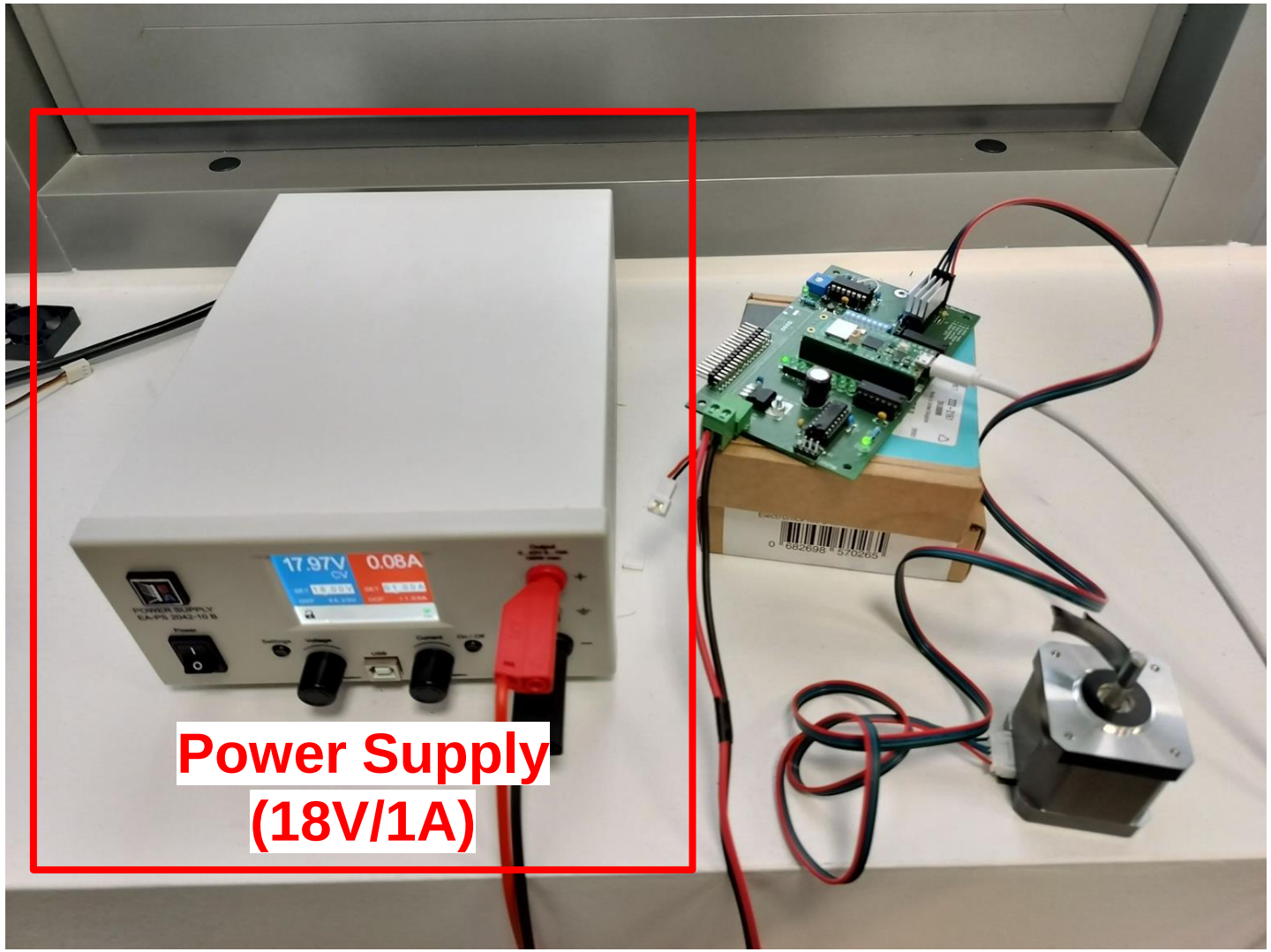
Test setup

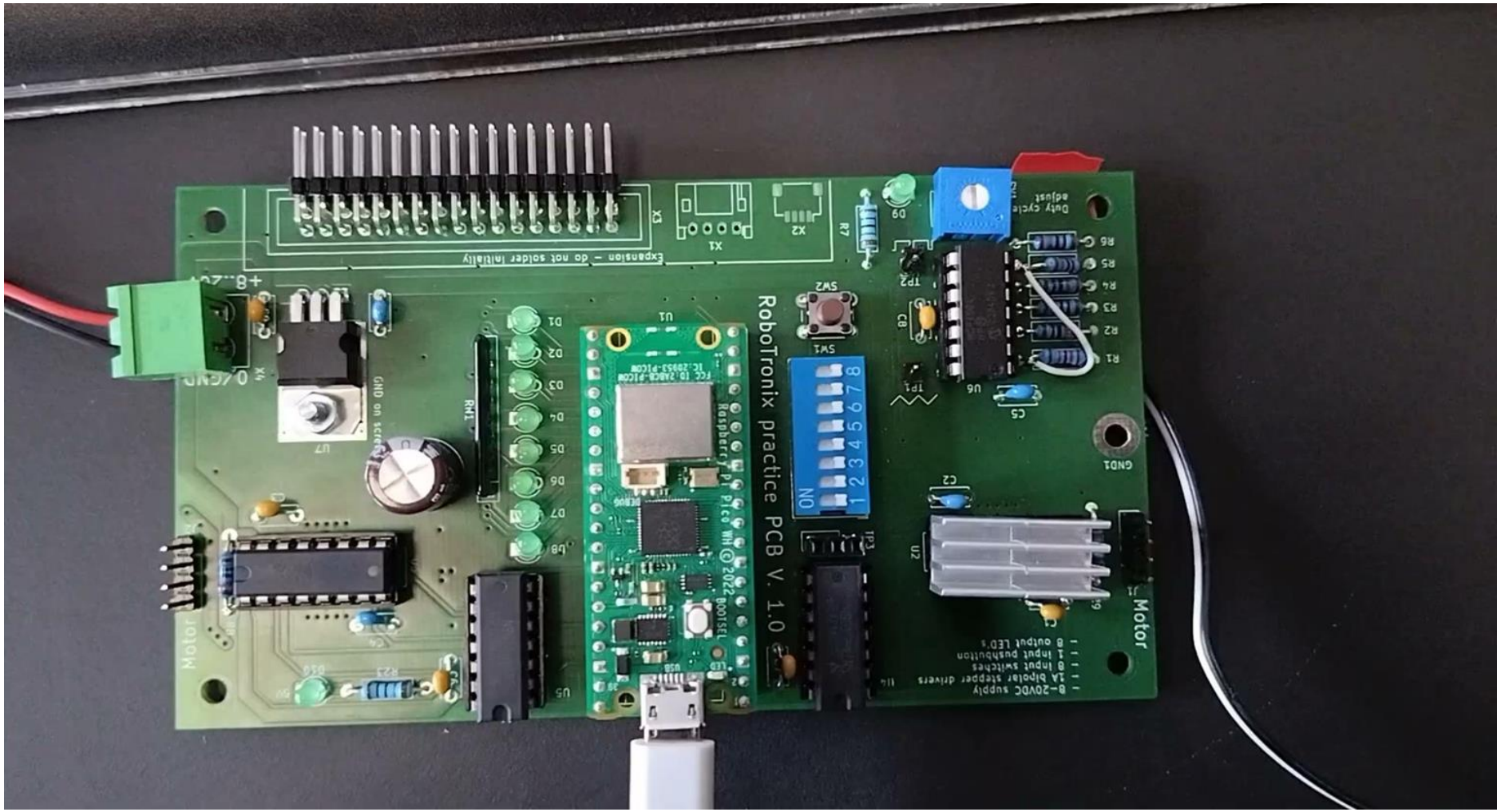


Test setup

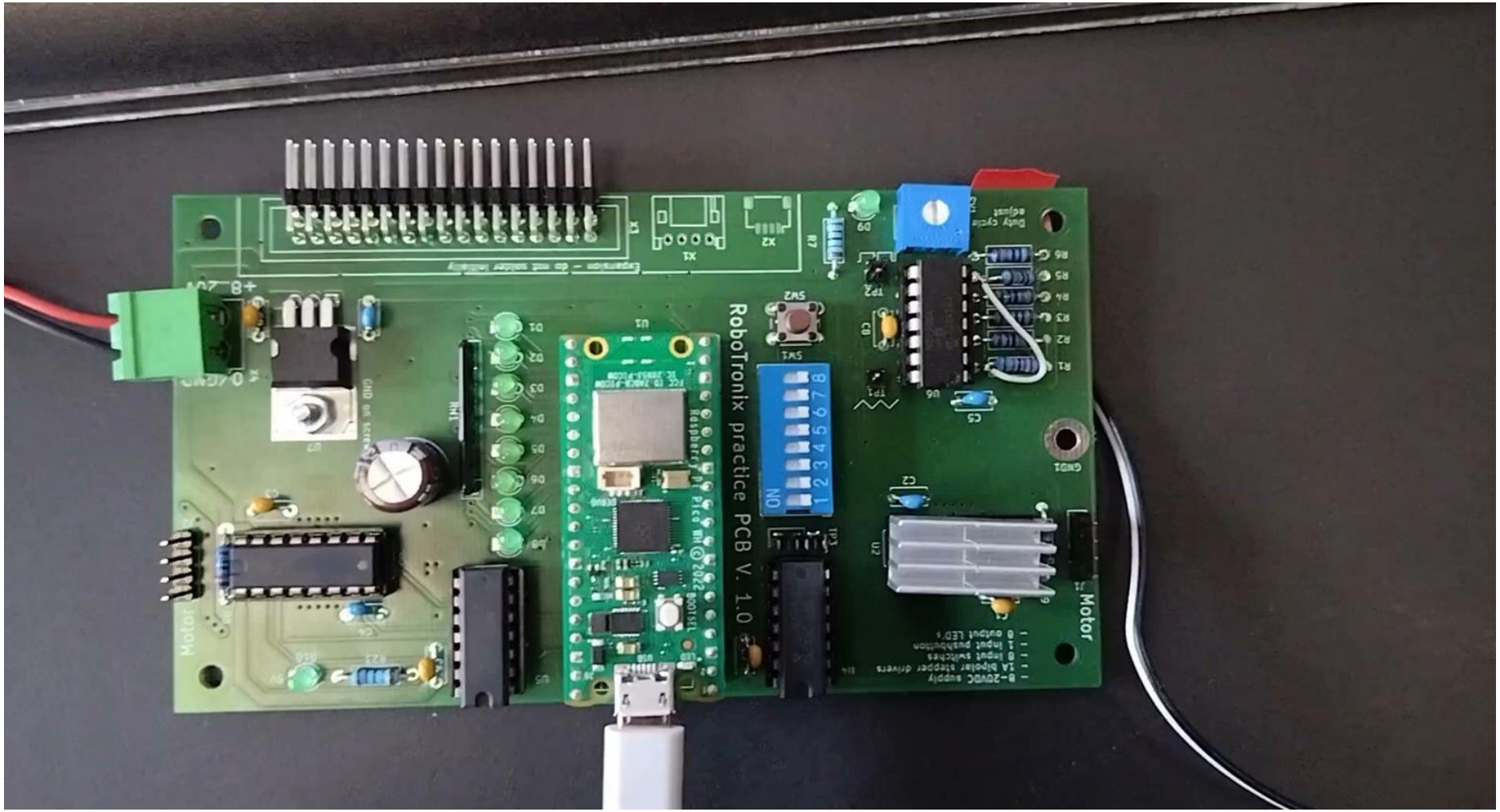


Test setup

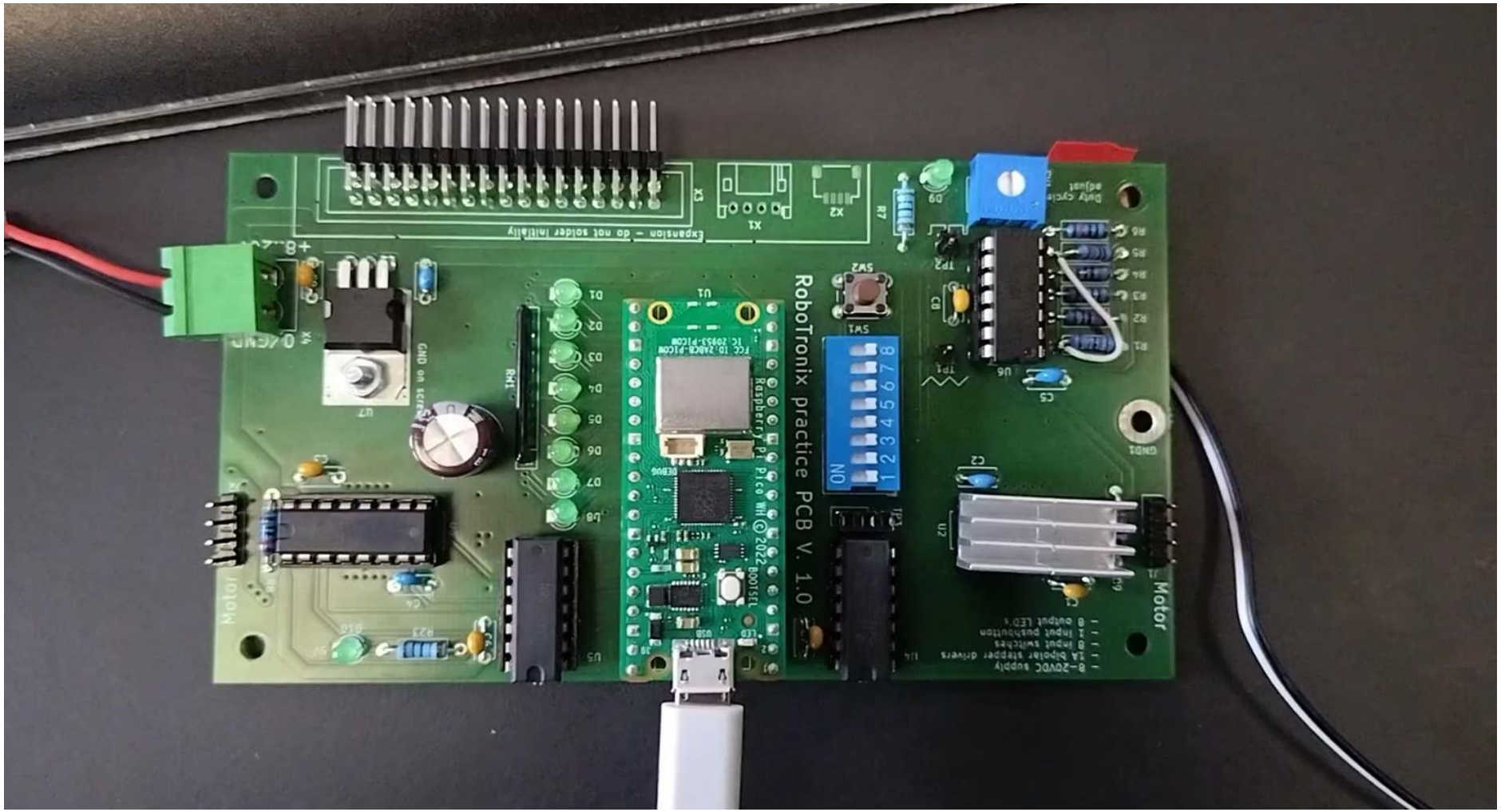




Full stepper: https://photos.app.goo.gl/hCaH4DwYtxyzFBie9



Half stepper: https://photos.app.goo.gl/WK56r75MM9Zwzdr16



Micro stepper: https://photos.app.goo.gl/UL5M3smoSGfwaNX97 (solution)

Demo using Thonny

Assignments