Vention Python API

Version 1.13.2

Contents

MachineLogic Python Programming v1.13.2

Interface

Compatibility

Vention's previous Python API User Manuals can be found in the 'Documentation for Previous Releases' at the bottom of this page.

This table specifies the compatibility of the Python package versions with the Vention's MachineMotion and Pendant versions.

Package	MachineMotion	Pendant
v1.12.x*	v2.13.x	v3.2, v3.3
v1.13.0	>=v2.14.x	>=v3.4
v1.13.1	>=v2.15.x	>=v3.5
v1.13.2	>=v2.15.x	>=v3.5

*Package versions <1.13.0 were published to PyPi under the package name machine-code-python-sdk

Documentation for Previous Releases

Vention Python API v.1.13.1 & Older

Vention Python API V1.13.1 User Manual
Vention Python API V1.13.0 User Manual
Vention Python API V1.12.1 User Manual

Change Log

[1.13.2] - 2024-11-11

Added

- Added robot async move capabilities with the addition of the following methods accessible off the Robot class: execute_sequence_async, movej_async, movel_async, wait_for_motion_async. Also added the following robot state: move_in_progress.

Changed:

- Updated documentation to reflect the new async methods and state.

[1.13.1] - 2024-07-18

Added

- Introduced the new Scene class to the SDK, accessible off ofMachine class by a new methodget_scene(). Scene is a software representation of the scene containing assets such as reference frames and targets for robots as defined within the MachineLogic Scene Assets pane.
- Added get_calibration_frame method to the newScene class which returns a software representation of a Calibration Frame (CalibrationFrame) as defined within the scene assets pane.
- Added get_default_value, get_calibrated_value, and set_calibrated_value to the newCalibrationFrame class, allowing users to calibrate their robot programmatically.

Changed

- Updated the documentation to reflect the new classesScene, and CalibrationFrame, along with their respective methods: get_calibration_frame (Scene), get_default_value (CalibrationFrame), get_calibrated_value(CalibrationFrame), and set_calibrated_value (CalibrationFrame).
- New compatibility matrix in documentation to define compatibilities between SDK versions, MachineMotion versions and Pendant versions.

[1.13.0] - 2024-04-28

Added

- Made ip_address optional in Machine class constructor to allow for more flexible deployment configurations.
- New PathFollower class implemented and importable through machinelogic. A Path Follower Object is a group of Actuators, Digital Inputs and Digital Outputs that enable execution of smooth predefined paths.
  These paths are defined with G-Code instructions. See Vention's G-code interface documentation for a list of supported commands:

Changed

- Updated documentation to reflect the new optionalip_address parameter in the Machine class.
- Modified RobotOperationalState by adding two new states: UNKNOWN = 4 and NEED_MANUAL_INTERVENTION = 5.
- Modified RobotSafetyState by removing: PROTECTIVE_STOP = 4 and adding: SAFEGUARD_STOP = 4.

[1.12.1] - 2023-02-28

- Initial python package release

Motion Features

When the Python program ends, any motion that is still executing will continue their execution. If you want to wait for a motion to complete, you should call:

```
actuator.wait_for_move_completion()
```

Asynchronous moves will not wait for the motion to complete before terminating the program.

The 'continuous_move' function will run forever if not stopped by the program.

Machine

A software representation of the entire Machine. A Machine is defined as any number of MachineMotions, each containing their own set of axes, outputs, inputs, pneumatics, bag grippers, and AC Motors. The Machine class offers a global way to retrieve these various components using the friendly names that you've defined in your MachineLogic configuration.

To create a new Machine with default settings, you can simply write:

```
machine = Machine()
```

If you need to connect to services running on a different machine or IP address, you can specify the IP address as follows:

```
machine = Machine("192.168.7.2")
```

You should only ever have a single instance of this object in your program.

get_ac_motor

- **Description** Retrieves an AC Motor by name.
  - **Parameters**
    - **name**
      - **Description** The name of the AC Motor.
      - **Type** str
  - **Returns**
    - **Description** The AC Motor that was found.
    - **Type** IACMotor
  - **Raises**
    - **Type** MachineMotionException
      - **Description** If it is not found.

get_actuator

- **Description** Retrieves an Actuator by name.
  - **Parameters**
    - **name**
      - **Description** The name of the Actuator.
      - **Type** str
  - **Returns**
    - **Description** The Actuator that was found.
    - **Type** IActuator
  - **Raises**
    - **Type** MachineException
      - **Description** If we cannot find the Actuator.

```
from machinelogic import Machine

machine = Machine()
my_actuator = machine.get_actuator("Actuator")


# Always home the actuator before starting to ensure position is properly calibrated.
my_actuator.home(timeout=10)

start_position = my_actuator.state.position
print("starting at position: ", start_position)

target_distance = 150.0  # mm

my_actuator.move_relative(
    distance=target_distance,
    timeout=10,  # seconds
)

# first_move_end_position is approx. equal to start_position + target_distance.
first_move_end_position = my_actuator.state.position
print("first move finished at position: ", first_move_end_position)


# move back to starting position
target_distance = -1 * target_distance
my_actuator.move_relative(
    distance=target_distance,
    timeout=10,  # seconds
)

# approx. equal to start_position,
end_position = my_actuator.state.position
print("finished back at position: ", end_position)
```

get_bag_gripper

- **Description** Retrieves a Bag Gripper by name.
  - **Parameters**
    - **name**
      - **Description** The name of the Bag Gripper
      - **Type** str
  - **Returns**
    - **Description** The Bag Gripper that was found.
    - **Type** IBagGripper
  - **Raises**
    - **Type** MachineMotionException
      - **Description** If it is not found.

```
from machinelogic import Machine

machine = Machine()
my_bag_gripper = machine.get_bag_gripper("Bag Gripper")
```

get_input

- **Description** Retrieves an DigitalInput by name.
  - **Parameters**
    - **name**
      - **Description** The name of the DigitalInput.
      - **Type** str
  - **Returns**
    - **Description** The DigitalInput that was found.
    - **Type** IDigitalInput
  - **Raises**
    - **Type** MachineException
      - **Description** If we cannot find the DigitalInput.

```
from machinelogic import Machine

machine = Machine()

my_input = machine.get_input("Input")

if my_input.state.value:
    print(f"{my_input.configuration.name} is HIGH")
else:
    print(f"{my_input.configuration.name} is LOW")
```

get_machine_motion

- **Description** Retrieves an IMachineMotion instance by name.
  - **Parameters**
    - **name**
      - **Description** The name of the MachineMotion.
      - **Type** str
  - **Returns**
    - **Description** The MachineMotion that was found.
    - **Type** IMachineMotion
  - **Raises**
    - **Type** MachineException
      - **Description** If we cannot find the MachineMotion.

```
from machinelogic import Machine, MachineException

machine = Machine()

my_controller_1 = machine.get_machine_motion("Controller 1")

configuration = my_controller_1.configuration

print("Name:", configuration.name)
print("IP Address:", configuration.ip_address)
```

get_output

- **Description** Retrieves an Output by name.
  - **Parameters**
    - **name**
      - **Description** The name of the Output
      - **Type** str
  - **Returns**
    - **Description** The Output that was found.
    - **Type** IOutput
  - **Raises**
    - **Type** MachineException
      - **Description** If we cannot find the Output.

```
from machinelogic import Machine, MachineException, DigitalOutputException

machine = Machine()
my_output = machine.get_output("Output")

my_output.write(True)  # Write "true" to the Output
my_output.write(False)  # Write "false" to the Output
```

get_pneumatic

- **Description** Retrieves a Pneumatic by name.
  - **Parameters**
    - **name**
      - **Description** The name of the Pneumatic.
      - **Type** str
  - **Returns**
    - **Description** The Pneumatic that was found.
    - **Type** IPneumatic
  - **Raises**
    - **Type** MachineException
      - **Description** If we cannot find the Pneumatic.

```
import time
from machinelogic import Machine

machine = Machine()
my_pneumatic = machine.get_pneumatic("Pneumatic")


# Idle
my_pneumatic.idle_async()
time.sleep(1)

# Push
my_pneumatic.push_async()
time.sleep(1)

# Pull
my_pneumatic.pull_async()
time.sleep(1)
```

get_robot

- **Description** Retrieves a Robot by name. If no name is specified, then returns the first Robot.
  - **Parameters**
    - **name**
      - **Description** The Robot name. If it's None, then the first Robot in the Robot list is returned.
      - **Type** str
  - **Returns**
    - **Description** The Robot that was found.
    - **Type** IRobot
  - **Raises**
    - **Type** MachineException
      - **Description** If the Robot is not found.

```
from machinelogic import Machine

machine = Machine()
my_robot = machine.get_robot("Robot")

# Example of use: moving robot to specified joint angles in deg
my_robot.movej([0, -90, 120, -90, -90, 0])
```

get_scene

- **Description** Returns the scene instance
  - **Returns**
    - **Description** The instance of the scene containing the scene assets.
    - **Type** IScene
  - **Raises**
    - **Type** MachineException
      - **Description** If failed to find the scene

on_mqtt_event

- **Description** Attach a callback function to an MQTT topic.
  - **Parameters**
    - **topic**
      - **Description** The topic to listen on.
      - **Type** str
    - **callback**
      - **Description** A callback where the first argument is the topic and the second is the message.
      - **Type** Union[Callable[[str, str], None], None]

```
import time
from machinelogic import Machine


machine = Machine()

my_event_topic = "my_custom/event/topic"


# A "callback" function called everytime a new mqtt event on my_event_topic is received.
def event_callback(topic: str, message: str):
    print("new mqtt event:", topic, message)


machine.on_mqtt_event(my_event_topic, event_callback)
machine.publish_mqtt_event(my_event_topic, "my message")

time.sleep(2)

machine.on_mqtt_event(my_event_topic, None)  # remove the callback.
```

publish_mqtt_event

- **Description** Publish an MQTT event.
  - **Parameters**
    - **topic**
      - **Description** Topic to publish.
      - **Type** str
    - **message**
      - **Description** Optional message.
      - **Type** Optional[str]
      - **Default** None

```
import time
import json
from machinelogic import Machine

machine = Machine()

# Example for publishing a cycle-start and cycle-end topic and message
# to track application cycles in MachineAnalytics
cycle_start_topic = "application/cycle-start"
cycle_end_topic = "application/cycle-end"
cycle_message = {
    "applicationId":"My Python Application",
    "cycleId":"default"
    }
json_cycle_message = json.dumps(cycle_message)

while True:
    machine.publish_mqtt_event(cycle_start_topic, json_cycle_message)
    print("Cycle Start")
    time.sleep(5)
    machine.publish_mqtt_event(cycle_end_topic, json_cycle_message)
    time.sleep(1)
    print("Cycle end")
```

MachineMotion

A software representation of a MachineMotion controller. The MachineMotion is comprised of many actuators, inputs, outputs, pneumatics, ac motors, and bag grippers. It keeps a persistent connection to MQTT as well.

You should NEVER construct this object yourself. Instead, it is best to rely on the Machine instance to provide you with a list of the available MachineMotions.

configuration

- **Description** MachineMotionConfiguration: The representation of the configuration associated with this MachineMotion.

Actuator

A software representation of an Actuator. An Actuator is defined as a motorized axis that can move by discrete distances. It is not recommended that you construct this object yourself. Rather, you should query it from a Machine instance:

E.g.:

```
machine = Machine()
my_actuator = machine.get_actuator("Actuator")
```

In this example, “New actuator” is the friendly name assigned to the Actuator in the MachineLogic configuration page.

configuration

- **Description** ActuatorConfiguration: The representation of the configuration associated with this MachineMotion.

home

- **Description** Home the Actuator synchronously.
  - **Parameters**
    - **timeout**
      - **Description** The timeout in seconds.
      - **Type** float
  - **Raises**
    - **Type** ActuatorException
      - **Description** If the home was unsuccessful or request timed out.

```
from machinelogic import Machine

machine = Machine()
my_actuator = machine.get_actuator("Actuator")

my_actuator.home(timeout=10)
```

lock_brakes

- **Description** Locks the brakes on this Actuator.
  - **Raises**
    - **Type** ActuatorException
      - **Description** If the brakes failed to lock.

```
from machinelogic import Machine

machine = Machine()
my_actuator = machine.get_actuator("Actuator")

my_actuator.unlock_brakes()

# Home the actuator before starting to ensure position is properly calibrated
my_actuator.home(timeout=10)
my_actuator.move_relative(distance=100.0)

my_actuator.lock_brakes()

# This move will fail because the brakes are now locked.
my_actuator.move_relative(distance=-100.0)
```

move_absolute

- **Description** Moves absolute synchronously to the specified position.
  - **Parameters**
    - **position**

        - **Description** The position to move to.
        - **Type** float
    - **timeout**
        - **Description** The timeout in seconds.
        - **Type** float
- **Raises**
    - **Type** ActuatorException
        - **Description** If the move was unsuccessful.

```
from machinelogic import Machine

machine = Machine()
my_actuator = machine.get_actuator("Actuator")



# Always home the actuator before starting to ensure position is properly calibrated.
my_actuator.home(timeout=10)



my_actuator.move_absolute(
    position=150.0,  # millimeters
    timeout=10,  # seconds
)
```

move_absolute_async

- **Description** Moves absolute asynchronously.
    - **Parameters**
        - **position**
            - **Description** The position to move to.
            - **Type** float
    - **Raises**
        - **Type** ActuatorException
            - **Description** If the move was unsuccessful.

```
import time
from machinelogic import Machine

machine = Machine()
my_actuator = machine.get_actuator("Actuator")

# Always home the actuator before starting to ensure position is properly calibrated.
my_actuator.home(timeout=10)

target_position = 150.0  # millimeters

# move_*_async will start the move and return without waiting for the move to complete.
my_actuator.move_absolute_async(target_position)

while my_actuator.state.move_in_progress:
    print("move is in progress...")
    time.sleep(1)

# end_position will be approx. equal to target_position.
end_position = my_actuator.state.position
print("finished at position: ", end_position)
```

move_continuous_async

- **Description** Starts a continuous move. The Actuator will keep moving until it is stopped.
  - **Parameters**
    - **speed**
      - **Description** The speed to move with.
      - **Type** float
      - **Default** 100.0
    - **acceleration**
      - **Description** The acceleration to move with.
      - **Type** float
      - **Default** 100.0
  - **Raises**
    - **Type** ActuatorException
      - **Description** If the move was unsuccessful.

```
import time
from machinelogic import Machine

machine = Machine()
my_actuator = machine.get_actuator("Actuator")

# Always home the actuator before starting to ensure position is properly calibrated.
my_actuator.home(timeout=10)

target_speed = 100.0  # mm/s
target_acceleration = 500.0  # mm/s^2
target_deceleration = 600.0  # mm/s^2

# move_*_async will start the move and return without waiting for the move to complete.
my_actuator.move_continuous_async(target_speed, target_acceleration)

time.sleep(10)  # move continuously for ~10 seconds.

my_actuator.stop(target_deceleration)  # decelerate to stopped.
```

move_relative

- **Description** Moves relative synchronously by the specified distance.
  - **Parameters**
    - **distance**
      - **Description** The distance to move.
      - **Type** float
    - **timeout**
      - **Description** The timeout in seconds.
      - **Type** float
  - **Raises**
    - **Type** ActuatorException
      - **Description** If the move was unsuccessful.

```
from machinelogic import Machine

machine = Machine()
my_actuator = machine.get_actuator("Actuator")


# Always home the actuator before starting to ensure position is properly calibrated.
my_actuator.home(timeout=10)

start_position = my_actuator.state.position
print("starting at position: ", start_position)

target_distance = 150.0  # mm

my_actuator.move_relative(
    distance=target_distance,
    timeout=10,  # seconds
)

# first_move_end_position is approx. equal to start_position + target_distance.
first_move_end_position = my_actuator.state.position
print("first move finished at position: ", first_move_end_position)


# move back to starting position
target_distance = -1 * target_distance
my_actuator.move_relative(
    distance=target_distance,
    timeout=10,  # seconds
)

# approx. equal to start_position,
end_position = my_actuator.state.position
print("finished back at position: ", end_position)
```

move_relative_async

- **Description** Moves relative asynchronously by the specified distance.
  - **Parameters**
    - **distance**
      - **Description** The distance to move.
      - **Type** float
  - **Raises**
    - **Type** ActuatorException
      - **Description** If the move was unsuccessful.

```
import time
from machinelogic import Machine

machine = Machine()
my_actuator = machine.get_actuator("Actuator")

# Always home the actuator before starting to ensure position is properly calibrated.
my_actuator.home(timeout=10)

start_position = my_actuator.state.position
print("starting at position: ", start_position)

target_distance = 150.0  # mm

# move_*_async will start the move and return without waiting for the move to complete.
my_actuator.move_relative_async(distance=150.0)

while my_actuator.state.move_in_progress:
    print("move is in progress...")
    time.sleep(1)

# end_position will be approx. equal to start_position + target_distance.
end_position = actuator.state.position
print("finished at position", end_position)
```

set_acceleration

- **Description** Sets the max acceleration for the Actuator.
  - **Parameters**
    - **acceleration**
      - **Description** The new acceleration.
      - **Type** float
  - **Raises**
    - **Type** ActuatorException
      - **Description** If the request was unsuccessful.

```
from machinelogic import Machine

machine = Machine()
my_actuator = machine.get_actuator("Actuator")

target_speed = 100.0  # mm/s
target_acceleration = 500.0  # mm/s^2

my_actuator.set_speed(target_speed)
my_actuator.set_acceleration(target_acceleration)
```

set_speed

- **Description** Sets the max speed for the Actuator.
  - **Parameters**
    - **speed**
      - **Description** The new speed.
      - **Type** float
  - **Raises**
    - **Type** ActuatorException
      - **Description** If the request was unsuccessful.

```
from machinelogic import Machine

machine = Machine()
my_actuator = machine.get_actuator("Actuator")

target_speed = 100.0  # mm/s
target_acceleration = 500.0  # mm/s^2

my_actuator.set_speed(target_speed)
my_actuator.set_acceleration(target_acceleration)
```

state

- **Description** ActuatorState: The representation of the current state of this MachineMotion.

stop

- **Description** Stops movement on this Actuator. If no argument is provided, then a quickstop is emitted which will abruptly stop the motion. Otherwise, the actuator will decelerate following the provided acceleration.
  - **Parameters**
    - **acceleration**
      - **Description** Deceleration speed.
      - **Type** float
  - **Raises**
    - **Type** ActuatorException
      - **Description** If the Actuator failed to stop.

```
from machinelogic import Machine

machine = Machine()
my_actuator = machine.get_actuator("Actuator")

deceleration = 500  # mm/s^2
my_actuator.stop(deceleration)  # Deceleration is an optional parameter
# The actuator will stop as quickly as possible if no deceleration is specified.
```

unlock_brakes

- **Description** Unlocks the brakes on this Actuator.

- **Raises**
  - **Type** ActuatorException
    - **Description** If the brakes failed to unlock.

```
from machinelogic import Machine

machine = Machine()
my_actuator = machine.get_actuator("Actuator")

my_actuator.unlock_brakes()

# Home the actuator before starting to ensure position is properly calibrated
my_actuator.home(timeout=10)
my_actuator.move_relative(distance=100.0)

my_actuator.lock_brakes()

# This move will fail because the brakes are now locked.
my_actuator.move_relative(distance=-100.0)
```

wait_for_move_completion

- **Description** Waits for motion to complete before commencing the next action.
  - **Parameters**
    - **timeout**
      - **Description** The timeout in seconds, after which an exception will be thrown.
      - **Type** float
  - **Raises**
    - **Type** ActuatorException
      - **Description** If the request fails or the move did not complete in the allocated amount of time.

```
from machinelogic import Machine

machine = Machine()
my_actuator = machine.get_actuator("Actuator")

# Always home the actuator before starting to ensure position is properly calibrated.
my_actuator.home(timeout=10)


target_position = 150.0  # millimeters
# move_*_async will start the move and return without waiting for the move to complete.
my_actuator.move_absolute_async(target_position)


print("move started...")
my_actuator.wait_for_move_completion(timeout=10)
print("motion complete.")


# end_position will be approx. equal to target_position.
end_position = my_actuator.state.position
print("finished at position: ", end_position)
```

ActuatorState

Representation of the current state of an Actuator instance. The values in this class are updated in real time to match the physical reality of the Actuator.

brakes

- **Description** float: The current state of the brakes of the Actuator. Set to 1 if locked, otherwise 0.

```
from machinelogic import Machine

machine = Machine()
my_actuator = machine.get_actuator("Actuator")
print(my_actuator.state.brakes)
```

end_sensors

- **Description** Tuple[bool, bool]: A tuple representing the state of the [home, end] sensors.

```
from machinelogic import Machine

machine = Machine()
my_actuator = machine.get_actuator("Actuator")
print(my_actuator.state.end_sensors)
```

move_in_progress

- **Description** bool: The boolean is True if a move is in progress, otherwise False.

```
from machinelogic import Machine

machine = Machine()
my_actuator = machine.get_actuator("Actuator")
print(my_actuator.state.move_in_progress)
```

output_torque

- **Description** dict[str, float]: The current torque output of the Actuator.

```
from machinelogic import Machine

machine = Machine()
my_actuator = machine.get_actuator("Actuator")
print(my_actuator.state.output_torque)
```

position

- **Description** float: The current position of the Actuator.

```
from machinelogic import Machine

machine = Machine()
my_actuator = machine.get_actuator("Actuator")
print(my_actuator.state.position)
```

speed

- **Description** float: The current speed of the Actuator.

```
from machinelogic import Machine

machine = Machine()
my_actuator = machine.get_actuator("Actuator")
print(my_actuator.state.speed)
```

ActuatorConfiguration

Representation of the configuration of an Actuator instance. This configuration defines what your Actuator is and how it should behave when work is requested from it.

actuator_type

- **Description** ActuatorType: The type of the Actuator.

```
from machinelogic import Machine

machine = Machine()
my_actuator = machine.get_actuator("Actuator")
print(my_actuator.configuration.actuator_type)
```

controller_id

- **Description** str: The controller id of the Actuator

home_sensor

- **Description** Literal[“A”, “B”]: The home sensor port, either A or B.

```
from machinelogic import Machine

machine = Machine()
my_actuator = machine.get_actuator("Actuator")
print(my_actuator.configuration.home_sensor)
```

ip_address

- **Description** str: The IP address of the Actuator.

```
from machinelogic import Machine

machine = Machine()
my_actuator = machine.get_actuator("Actuator")
print(my_actuator.configuration.ip_address)
```

name

- **Description** str: The name of the Actuator.

```
from machinelogic import Machine

machine = Machine()
my_actuator = machine.get_actuator("Actuator")
print(my_actuator.configuration.name)
```

units

- **Description** Literal[“deg”, “mm”]: The units that the Actuator functions in.

```
from machinelogic import Machine

machine = Machine()
my_actuator = machine.get_actuator("Actuator")
print(my_actuator.configuration.units)
```

uuid

- **Description** str: The Actuator's ID.

```
from machinelogic import Machine

machine = Machine()
my_actuator = machine.get_actuator("Actuator")
print(my_actuator.configuration.uuid)
```

ActuatorGroup

A helper class used to group N-many Actuator instances together so that they can be acted upon as a group. An ActuatorGroup may only contain Actuators that are on the same MachineMotion controller.

E.g.:

```
machine = Machine()
my_actuator_1 = machine.get_actuator("Actuator 1")
my_actuator_2 = machine.get_actuator("Actuator 2")
actuator_group = ActuatorGroup(my_actuator_1, my_actuator_2)
```

lock_brakes

- **Description** Locks the brakes for all Actuators in the group.
  - **Raises**
    - **Type** ActuatorGroupException
      - **Description** If the brakes failed to lock on a single Actuator in the group.

```
from machinelogic import Machine, ActuatorGroup

machine = Machine()
my_actuator_1 = machine.get_actuator("Actuator 1")
my_actuator_2 = machine.get_actuator("Actuator 2")

# Always home the actuators before starting to ensure position is properly calibrated.
my_actuator_1.home(timeout=10)
my_actuator_2.home(timeout=10)

actuator_group = ActuatorGroup(my_actuator_1, my_actuator_2)
actuator_group.set_speed(100.0)  # mm/s
actuator_group.set_acceleration(500.0)  # mm/s^2

actuator_group.lock_brakes()

# This move will fail because the brakes are locked.
actuator_group.move_absolute((50.0, 120.0), timeout=10)
```

move_absolute

- **Description** Moves absolute synchronously to the tuple of positions.
  - **Parameters**
    - **position**
      - **Description** The positions to move to. Each value corresponds 1-to-1 with the actuators tuple provided to the constructor.
      - **Type** Tuple[float, …]
    - **timeout**
      - **Description** The timeout in seconds after which an exception is thrown.
      - **Type** float
      - **Default** DEFAULT_MOVEMENT_TIMEOUT_SECONDS
  - **Raises**
    - **Type** ActuatorGroupException
      - **Description** If the request fails or the timeout occurs.

```
from machinelogic import Machine, ActuatorGroup

machine = Machine()
my_actuator_1 = machine.get_actuator("Actuator 1")
my_actuator_2 = machine.get_actuator("Actuator 2")

# Always home the actuators before starting to ensure position is properly calibrated.
my_actuator_1.home(timeout=10)
my_actuator_2.home(timeout=10)

actuator_group = ActuatorGroup(my_actuator_1, my_actuator_2)
actuator_group.set_speed(100.0)  # mm/s
actuator_group.set_acceleration(500.0)  # mm/s^2


target_positions = (100.0, 200.0)  # (mm - actuator1, mm - actuator2)

actuator_group.move_absolute(target_positions, timeout=10)
```

move_absolute_async

- **Description** Moves absolute asynchronously to the tuple of positions.
  - **Parameters**
    - **distance**
      - **Description** The positions to move to. Each value corresponds 1-to-1 with the actuators tuple provided to the constructor.
      - **Type** Tuple[float, …]
  - **Raises**
    - **Type** ActuatorGroupException
      - **Description** If the request fails.

```
from machinelogic import Machine, ActuatorGroup

machine = Machine()
my_actuator_1 = machine.get_actuator("Actuator 1")
my_actuator_2 = machine.get_actuator("Actuator 2")

# Always home the actuators before starting to ensure position is properly calibrated.
my_actuator_1.home(timeout=10)
my_actuator_2.home(timeout=10)

actuator_group = ActuatorGroup(my_actuator_1, my_actuator_2)
actuator_group.set_speed(100.0)  # mm/s
actuator_group.set_acceleration(500.0)  # mm/s^2

target_positions = (75.0, 158.0)  # (mm - actuator1, mm - actuator2)

# move_*_async will start the move and return without waiting for the move to complete.
actuator_group.move_absolute_async(target_positions)
print("move started..")

actuator_group.wait_for_move_completion()
print("motion completed.")
```

move_relative

- **Description** Moves relative synchronously by the tuple of distances.
  - **Parameters**
    - **distance**
      - **Description** The distances to move each Actuator. Each value corresponds 1-to-1 with the actuators tuple provided to the constructor.
      - **Type** Tuple[float, …]
    - **timeout**
      - **Description** The timeout in seconds after which an exception is thrown.
      - **Type** float
      - **Default** DEFAULT_MOVEMENT_TIMEOUT_SECONDS
  - **Raises**
    - **Type** ActuatorGroupException
      - **Description** If the request fails or the timeout occurs

```
from machinelogic import Machine, ActuatorGroup

machine = Machine()
my_actuator_1 = machine.get_actuator("Actuator 1")
my_actuator_2 = machine.get_actuator("Actuator 2")

# Always home the actuators before starting to ensure position is properly calibrated.
my_actuator_1.home(timeout=10)
my_actuator_2.home(timeout=10)

actuator_group = ActuatorGroup(my_actuator_1, my_actuator_2)
actuator_group.set_speed(100.0)  # mm/s
actuator_group.set_acceleration(500.0)  # mm/s^2


target_distances = (-120.0, 240.0)  # (mm - actuator1, mm - actuator2)

actuator_group.move_relative(target_distances, timeout=10)
```

move_relative_async

- **Description** Moves relative asynchronously by the tuple of distances.
  - **Parameters**
    - **distance**
      - **Description** The distances to move each Actuator. Each value corresponds 1-to-1 with the actuators tuple provided to the constructor.
      - **Type** Tuple[float, …]
  - **Raises**
    - **Type** ActuatorGroupException
      - **Description** If the request fails.

```
import time
from machinelogic import Machine, ActuatorGroup

machine = Machine()
actuator1 = machine.get_actuator("My Actuator #1")
actuator2 = machine.get_actuator("My Actuator #2")

# Always home the actuators before starting to ensure position is properly calibrated.
actuator1.home(timeout=10)
actuator2.home(timeout=10)

actuator_group = ActuatorGroup(actuator1, actuator2)
actuator_group.set_speed(100.0)  # mm/s
actuator_group.set_acceleration(500.0)  # mm/s^2


target_distances = (-120.0, 240.0)  # (mm - actuator1, mm - actuator2)

# move_*_async will start the move and return without waiting for the move to complete.
actuator_group.move_relative_async(target_distances)

while actuator_group.state.move_in_progress:
    print("motion is in progress..")
    time.sleep(1)

print("motion complete")
```

set_acceleration

- **Description** Sets the acceleration on all Actuators in the group.
  - **Parameters**
    - **acceleration**
      - **Description** The acceleration to set on all Actuators in the group.
      - **Type** float
  - **Raises**
    - **Type** ActuatorGroupException
      - **Description** If the acceleration failed to set on any Actuator in the group.

set_speed

- **Description** Sets the speed on all Actuators in the group.
  - **Parameters**
    - **speed**
      - **Description** The speed to set on all Actuators in the group.
      - **Type** float
  - **Raises**
    - **Type** ActuatorGroupException
      - **Description** If the speed failed to set on any Actuator in the group.

state

- **Description** ActuatorGroupState: The state of the ActuatorGroup.

stop

- **Description** Stops movement on all Actuators in the group.
  - **Raises**
    - **Type** ActuatorGroupException
      - **Description** If any of the Actuators in the group failed to stop.

unlock_brakes

- **Description** Unlocks the brakes on all Actuators in the group.
  - **Raises**
    - **Type** ActuatorGroupException
      - **Description** If the brakes failed to unlock on a single Actuator in the group.

```
from machinelogic import Machine, ActuatorGroup

machine = Machine()
my_actuator_1 = machine.get_actuator("Actuator 1")
my_actuator_2 = machine.get_actuator("Actuator 2")

# Always home the actuators before starting to ensure position is properly calibrated.
my_actuator_1.home(timeout=10)
my_actuator_2.home(timeout=10)

actuator_group = ActuatorGroup(my_actuator_1, my_actuator_2)
actuator_group.set_speed(100.0)  # mm/s
actuator_group.set_acceleration(500.0)  # mm/s^2

actuator_group.lock_brakes()

# This move will fail because the brakes are locked.
actuator_group.move_absolute((50.0, 120.0), timeout=10)
```

wait_for_move_completion

- **Description** Waits for motion to complete on all Actuators in the group.
  - **Parameters**
    - **timeout**
      - **Description** The timeout in seconds, after which an exception will be thrown.
      - **Type** float
  - **Raises**
    - **Type** ActuatorGroupException
      - **Description** If the request fails or the move did not complete in the allocated amount of time.

```
from machinelogic import Machine, ActuatorGroup

machine = Machine()
my_actuator_1 = machine.get_actuator("Actuator 1")
my_actuator_2 = machine.get_actuator("Actuator 2")

# Always home the actuators before starting to ensure position is properly calibrated.
my_actuator_1.home(timeout=10)
my_actuator_2.home(timeout=10)

actuator_group = ActuatorGroup(my_actuator_1, my_actuator_2)
actuator_group.set_speed(100.0)  # mm/s
actuator_group.set_acceleration(500.0)  # mm/s^2

target_positions = (75.0, 158.0)  # (mm - actuator1, mm - actuator2)

# move_*_async will start the move and return without waiting for the move to complete.
actuator_group.move_absolute_async(target_positions)
print("move started..")

actuator_group.wait_for_move_completion()
print("motion completed.")
```

Robot

A software representation of a Robot. It is not recommended that you construct this object yourself. Rather, you should query it from a Machine instance:

E.g.:

```
machine = Machine()
my_robot = machine.get_robot("Robot")
```

In this example, “Robot” is the friendly name assigned to the actuator in the MachineLogic configuration page.

compute_forward_kinematics

- **Description** Computes the forward kinematics from joint angles.
  - **Parameters**
    - **joint_angles**
      - **Description** The 6 joint angles, in degrees.
      - **Type** JointAnglesDegrees
  - **Returns**
    - **Description** Cartesian pose, in mm and degrees
    - **Type** CartesianPose
  - **Raises**
    - **Type** ValueError
      - **Description** Throws an error if the joint angles are invalid.

```
from machinelogic import Machine

machine = Machine()
my_robot = machine.get_robot("Robot")

# Joint angles, in degrees
joint_angles = [
    176.68,   # j1
    -35.95,   # j2
     86.37,   # j3
   -150.02,   # j4
    -90.95,   # j5
    -18.58,   # j6
]
computed_robot_pose = my_robot.compute_forward_kinematics(joint_angles)
print(computed_robot_pose)
```

compute_inverse_kinematics

- **Description** Computes the inverse kinematics from a Cartesian pose.
  - **Parameters**
    - **cartesian_position**
      - **Description** The end effector's pose, in mm and degrees, where the angles are extrinsic Euler angles in XYZ order.
      - **Type** CartesianPose
  - **Returns**
    - **Description** Joint angles, in degrees.
    - **Type** JointAnglesDegrees
  - **Raises**
    - **Type** ValueError
      - **Description** Throws an error if the inverse kinematic solver fails.

```
from machinelogic import Machine

machine = Machine()
my_robot = machine.get_robot("Robot")

cartesian_position = [
     648.71,  # x in millimeters
    -313.30,  # y in millimeters
     159.28,  # z in millimeters
     107.14,  # rx in degrees
    -145.87,  # ry in degrees
      15.13,  # rz in degrees
]

computed_joint_angles = my_robot.compute_inverse_kinematics(cartesian_position)
print(computed_joint_angles)
```

configuration

- **Description** The Robot configuration.

create_sequence

- **Description** Creates a sequence-builder object for building a sequence of robot movements. This method is expected to be used with the append_*
methods.
  - **Returns**
    - **Description** A sequence builder object.
    - **Type** SequenceBuilder

```
from machinelogic import Machine

machine = Machine()
my_robot = machine.get_robot("Robot")

# Create an arbitrary Cartesian waypoint, that is 10mm or 10 degrees away from the current position
cartesian_waypoint = [i + 10 for i in my_robot.state.cartesian_position]

# Create an arbitrary joint waypoint, that is 10 degrees away from the current joint angles
joint_waypoint = [i + 10 for i in my_robot.state.joint_angles]

cartesian_velocity = 100.0  # millimeters per second
cartesian_acceleration = 100.0  # millimeters per second squared
blend_factor_1 = 0.5

joint_velocity = 10.0  # degrees per second
joint_acceleration = 10.0  # degrees per second squared
blend_factor_2 = 0.5


with my_robot.create_sequence() as seq:
    seq.append_movel(
        cartesian_waypoint, cartesian_velocity, cartesian_acceleration, blend_factor_1
    )
    seq.append_movej(joint_waypoint, joint_velocity, joint_acceleration, blend_factor_2)

# Alternate Form:
seq = my_robot.create_sequence()
seq.append_movel(cartesian_waypoint)
seq.append_movej(joint_waypoint)
my_robot.execute_sequence(seq)
```

execute_sequence

- **Description** Moves the robot through a specific sequence of joint and linear motions.
  - **Parameters**
    - **sequence**
      - **Description** The sequence of target points.

        - **Type** SequenceBuilder
    - **Returns**
        - **Description** True if successful.
        - **Type** bool

```
from machinelogic import Machine

machine = Machine()
my_robot = machine.get_robot("Robot")

# Create an arbitrary Cartesian waypoint, that is 10mm or 10 degrees away from the current position
cartesian_waypoint = [i + 10 for i in my_robot.state.cartesian_position]

# Create an arbitrary joint waypoint, that is 10 degrees away from the current joint angles
joint_waypoint = [i + 10 for i in my_robot.state.joint_angles]

cartesian_velocity = 100.0  # millimeters per second
cartesian_acceleration = 100.0  # millimeters per second squared
blend_factor_1 = 0.5

joint_velocity = 10.0  # degrees per second
joint_acceleration = 10.0  # degrees per second squared
blend_factor_2 = 0.5

seq = my_robot.create_sequence()
seq.append_movel(
    cartesian_waypoint, cartesian_velocity, cartesian_acceleration, blend_factor_1
)
seq.append_movej(joint_waypoint, joint_velocity, joint_acceleration, blend_factor_2)
my_robot.execute_sequence(seq)
```

execute_sequence_async

- **Description** Moves the robot through a specific sequence of joint and linear motions asynchronously.
    - **Parameters**
        - **sequence**
            - **Description** The sequence of target points.
            - **Type** SequenceBuilder
    - **Returns**
        - **Description** True if successful.
        - **Type** bool

```
from machinelogic import Machine

machine = Machine()
my_robot = machine.get_robot("Robot")

# Create an arbitrary Cartesian waypoint, that is 10mm or 10 degrees away from the current position
cartesian_waypoint = [i + 10 for i in my_robot.state.cartesian_position]

# Create an arbitrary joint waypoint, that is 10 degrees away from the current joint angles
joint_waypoint = [i + 10 for i in my_robot.state.joint_angles]

cartesian_velocity = 100.0  # millimeters per second
cartesian_acceleration = 100.0  # millimeters per second squared
blend_factor_1 = 0.5

joint_velocity = 10.0  # degrees per second
joint_acceleration = 10.0  # degrees per second squared
blend_factor_2 = 0.5

seq = my_robot.create_sequence()
seq.append_movel(
    cartesian_waypoint, cartesian_velocity, cartesian_acceleration, blend_factor_1
)
seq.append_movej(joint_waypoint, joint_velocity, joint_acceleration, blend_factor_2)

my_robot.execute_sequence_async(seq)

print("Robot is executing sequence asynchronously.")
my_robot.wait_for_motion_completion()
print("Robot has finished executing sequence.")
```

move_stop

- **Description** Stops the robot current movement.
  - **Returns**
    - **Description** True if the robot was successfully stopped, False otherwise.
    - **Type** bool

```
from machinelogic import Machine

machine = Machine()
my_robot = machine.get_robot("Robot")

my_robot.move_stop()
```

movej

- **Description** Moves the robot to a specified joint position.
  - **Parameters**

- **target**
  - **Description** The target joint angles, in degrees.
  - **Type** JointAnglesDegrees
- **velocity**
  - **Description** The joint velocity to move at, in degrees per second.
  - **Type** DegreesPerSecond
- **acceleration**
  - **Description** The joint acceleration to move at, in degrees per second squared.
  - **Type** DegreesPerSecondSquared

```
from machinelogic import Machine

machine = Machine()
my_robot = machine.get_robot("Robot")

joint_velocity = 10.0  # degrees per second
joint_acceleration = 10.0  # degrees per second squared

# Joint angles, in degrees
joint_angles = [
    86.0,  # j1
     0.0,  # j2
    88.0,  # j3
     0.0,  # j4
    91.0,  # j5
     0.0,  # j6
]

my_robot.movej(
    joint_angles,
    joint_velocity,
    joint_acceleration,
)
```

movej_async

- **Description** Moves the robot to a specified joint position asynchronously.
  - **Parameters**
    - **target**
      - **Description** The target joint angles, in degrees.
      - **Type** JointAnglesDegrees
    - **velocity**
      - **Description** The joint velocity to move at, in degrees per second.
      - **Type** DegreesPerSecond
    - **acceleration**
      - **Description** The joint acceleration to move at, in degrees per second squared.
      - **Type** DegreesPerSecondSquared

```
from machinelogic import Machine

machine = Machine()
my_robot = machine.get_robot("Robot")

joint_velocity = 10.0  # degrees per second
joint_acceleration = 10.0  # degrees per second squared

# Joint angles, in degrees
joint_angles = [
    86.0,  # j1
     0.0,  # j2
    88.0,  # j3
     0.0,  # j4
    91.0,  # j5
     0.0,  # j6
]

my_robot.movej_async(
  joint_angles,
  joint_velocity,
  joint_acceleration,
)
print("Robot is moving asynchronously to the specified joint angles.")
my_robot.wait_for_motion_completion()
print("Robot has finished moving to the specified joint angles.")
```

movel

- **Description** Moves the robot to a specified Cartesian position.
  - **Parameters**
    - **target**
      - **Description** The end effector's pose, in mm and degrees, where the angles are extrinsic Euler angles in XYZ order.
      - **Type** CartesianPose
    - **velocity**
      - **Description** The velocity to move at, in mm/s.
      - **Type** MillimetersPerSecond
    - **acceleration**
      - **Description** The acceleration to move at, in mm/s^2.
      - **Type** MillimetersPerSecondSquared
    - **reference_frame**
      - **Description** The reference frame to move relative to. If None, the robot's base frame is used.
      - **Type** CartesianPose

```
from machinelogic import Machine

machine = Machine()
my_robot = machine.get_robot("Robot")

linear_velocity = 100.0  # millimeters per second
linear_acceleration = 100.0  # millimeters per second squared

# Target Cartesian pose, in millimeters and degrees
cartesian_pose = [
   -267.8,    # x in millimeters
    -89.2,    # y in millimeters
    277.4,    # z in millimeters
   -167.8,    # rx in degrees
        0,    # ry in degrees
    -77.8,    # rz in degrees
]

reference_frame = [
    23.56,    # x in millimeters
  -125.75,    # y in millimeters
     5.92,    # z in millimeters
     0.31,    # rx in degrees
     0.65,    # ry in degrees
    90.00,    # rz in degrees
]

my_robot.movel(
    cartesian_pose,
    linear_velocity,  # Optional
    linear_acceleration,  # Optional
    reference_frame,  # Optional
)
```

movel_async

- **Description** Moves the robot to a specified Cartesian position asynchronously.
  - **Parameters**
    - **target**
      - **Description** The end effector's pose, in mm and degrees, where the angles are extrinsic Euler angles in XYZ order.
      - **Type** CartesianPose
    - **velocity**
      - **Description** The velocity to move at, in mm/s.
      - **Type** MillimetersPerSecond
    - **acceleration**
      - **Description** The acceleration to move at, in mm/s^2.
      - **Type** MillimetersPerSecondSquared
    - **reference_frame**
      - **Description** The reference frame to move relative to. If None, the robot's base frame is used.
      - **Type** CartesianPose

```
from machinelogic import Machine

machine = Machine()
my_robot = machine.get_robot("Robot")

linear_velocity = 100.0  # millimeters per second
linear_acceleration = 100.0  # millimeters per second squared

# Target Cartesian pose, in millimeters and degrees
cartesian_pose = [
   -267.8,    # x in millimeters
    -89.2,    # y in millimeters
    277.4,    # z in millimeters
   -167.8,    # rx in degrees
        0,    # ry in degrees
    -77.8,    # rz in degrees
]

reference_frame = [
    23.56,    # x in millimeters
  -125.75,    # y in millimeters
     5.92,    # z in millimeters
     0.31,    # rx in degrees
     0.65,    # ry in degrees
    90.00,    # rz in degrees
]

my_robot.movel_async(
    cartesian_pose,
    linear_velocity,  # Optional
    linear_acceleration,  # Optional
    reference_frame,  # Optional
)
print("Robot is moving asynchronously to the specified Cartesian pose.")
my_robot.wait_for_motion_completion()
print("Robot has finished moving to the specified Cartesian pose.")
```

on_log_alarm

- **Description** Set a callback to the log alarm.
  - **Parameters**
    - **callback**
      - **Description** A callback function to be called when a robot alarm is received.
      - **Type** Callable[[IRobotAlarm], None]
  - **Returns**
    - **Description** The callback ID.
    - **Type** int

```
from machinelogic import Machine

machine = Machine()
my_robot = machine.get_robot("Robot")


# The functon defined here is called when the specified alarm occurs
def handle_log_alarm(alarm):
    print(alarm.level, alarm.error_code, alarm.description)


my_robot.on_log_alarm(handle_log_alarm)
```

on_system_state_change

- **Description** Registers a callback for system state changes.
  - **Parameters**
    - **callback**
      - **Description** The callback function.
      - **Type** Callable[[RobotOperationalState, RobotSafetyState], None]
  - **Returns**
    - **Description** The callback ID.
    - **Type** int

```
from machinelogic import Machine
from machinelogic.machinelogic.robot import RobotOperationalState, RobotSafetyState

machine = Machine()
my_robot = machine.get_robot("Robot")


# The function defined here is called when the specified state change occurs
def handle_state_change(robot_operational_state: RobotOperationalState, safety_state: RobotSafetyState):
    """
    A function that is called when the specified state change occurs.

    Args:
        robot_operational_state (RobotOperationalState): The current operational state of the robot.
        safety_state (RobotSafetyState): The current safety state of the robot.
    """
    print(robot_operational_state, safety_state)


callback_id = my_robot.on_system_state_change(handle_state_change)
print(callback_id)
```

reset

- **Description** Attempts to reset the robot to a normal operational state.

- **Returns**
  - **Description** True if successful.
  - **Type** bool

```
from machinelogic import Machine

machine = Machine()
my_robot = machine.get_robot("Robot")

did_reset = my_robot.reset()
print(did_reset)

# Robot state should be 'Normal'
print(my_robot.state.operational_state)
```

set_payload

- **Description** Sets the payload of the robot.
  - **Parameters**
    - **payload**
      - **Description** The payload, in kg.
      - **Type** Kilograms
  - **Returns**
    - **Description** True if successful.
    - **Type** bool

```
from machinelogic import Machine

machine = Machine()
my_robot = machine.get_robot("Robot")

# Weight in Kilograms
weight = 2.76
is_successful = my_robot.set_payload(weight)
print(is_successful)
```

set_tcp_offset

- **Description** Sets the tool center point offset.
  - **Parameters**
    - **tcp_offset**
      - **Description** The TCP offset, in mm and degrees, where the angles are extrinsic Euler angles in XYZ order.
      - **Type** CartesianPose
  - **Returns**
    - **Description** True if the TCP offset was successfully set, False otherwise.
    - **Type** bool

```
from machinelogic import Machine

machine = Machine()
my_robot = machine.get_robot("Robot")

# This offset will be applied in reference
# to the end effector coordinate system
cartesian_offset = [
    10.87,   # x in millimeters
   -15.75,   # y in millimeters
   200.56,   # z in millimeters
     0.31,   # rx degrees
     0.65,   # ry degrees
     0.00,   # rz degrees
]

is_successful = my_robot.set_tcp_offset(cartesian_offset)
print(is_successful)
```

set_tool_digital_output

- **Description** Sets the value of a tool digital output.
  - **Parameters**
    - **pin**
      - **Description** The pin number.
      - **Type** int
    - **value**
      - **Description** The value to set, where 1 is high and 0 is low.
      - **Type** int
  - **Returns**
    - **Description** True if successful.
    - **Type** bool

```
from machinelogic import Machine

machine = Machine()
# New robot must be configured in the Configuration pane
my_robot = machine.get_robot("Robot")

# digital output identifier
output_pin = 1
value = 0
is_successful = my_robot.set_tool_digital_output(output_pin, value)
print(is_successful)
```

state

- **Description** The current Robot state.

teach_mode

- **Description** Put the robot into teach mode (i.e., freedrive).
  - **Returns**
    - **Description** A context manager that will exit teach mode when it is closed.
    - **Type** TeachModeContextManager

```
import time

from machinelogic import Machine

machine = Machine()
my_robot = machine.get_robot("Robot")

with my_robot.teach_mode():  # When all arguments inside this statement are complete, teach mode ends automatically
    print("Robot is now in teach mode for 5 seconds")
    time.sleep(5)

    # Robot should be in 'Freedrive'
    print(my_robot.state.operational_state)
    time.sleep(1)

time.sleep(1)

# Robot should be back to 'Normal'
print(my_robot.state.operational_state)
```

wait_for_motion_completion

- **Description** Waits for the robot to complete its current motion. Used in asynchronous movements.
  - **Parameters**
    - **timeout**
      - **Description** The timeout in seconds, after which an exception will be thrown.
      - **Type** float
  - **Returns**
    - **Description** True if successful.
    - **Type** bool
  - **Raises**
    - **Type** ActuatorException
      - **Description** If the request fails or the move did not complete in the allocated amount of time.

```
from machinelogic import Machine

machine = Machine()
my_robot = machine.get_robot("Robot")

joint_velocity = 10.0  # degrees per second
joint_acceleration = 10.0  # degrees per second squared

# Joint angles, in degrees
joint_angles = [
    86.0,  # j1
     0.0,  # j2
    88.0,  # j3
     0.0,  # j4
    91.0,  # j5
     0.0,  # j6
]

my_robot.movej_async(
  joint_angles,
  joint_velocity,
  joint_acceleration,
)
print("Robot is moving asynchronously to the specified joint angles.")
my_robot.wait_for_motion_completion()
print("Robot has finished moving to the specified joint angles.")
```

RobotState

A representation of the robot current state.

cartesian_position

- **Description** The end effector's pose, in mm and degrees, where the angles are extrinsic Euler angles in XYZ order.

```
from machinelogic import Machine

machine = Machine()
my_robot = machine.get_robot("Robot")

end_effector_pose = my_robot.state.cartesian_position
end_effector_position_mm = end_effector_pose[:3]
end_effector_orientation_euler_xyz_deg = end_effector_pose[-3:]
print(f"End effector's pose: {end_effector_pose}")
print(f"End effector's Cartesian position: {end_effector_position_mm}")
print(f"End effector's Euler XYZ orientation: {end_effector_orientation_euler_xyz_deg}")
```

get_digital_input_value

- **Description** Returns the value of a digital input at a given pin.
  - **Parameters**
    - **pin**
      - **Description** The pin number.
      - **Type** int
  - **Returns**
    - **Description** True if the pin is high, False otherwise.
    - **Type** None

```
from machinelogic import Machine

machine = Machine()
my_robot = machine.get_robot("Robot")
print(my_robot.state.get_digital_input_value(0))
```

joint_angles

- **Description** The robot current joint angles.

```
from machinelogic import Machine

machine = Machine()
my_robot = machine.get_robot("Robot")

print(my_robot.state.joint_angles)
```

move_in_progress

- **Description** Check if the robot is currently moving.

```
from machinelogic import Machine

machine = Machine()
my_robot = machine.get_robot("Robot")
print(my_robot.state.move_in_progress)
```

operational_state

- **Description** The current robot operational state.

```
from machinelogic import Machine

machine = Machine()
my_robot = machine.get_robot("Robot")
print(my_robot.state.operational_state)
```

safety_state

- **Description** The current robot safety state.

```
from machinelogic import Machine

machine = Machine()
my_robot = machine.get_robot("Robot")
print(my_robot.state.safety_state)
```

RobotConfiguration

A representation of the configuration of a Robot instance. This configuration defines what your Robot is and how it should behave when work is requested from it.

cartesian_velocity_limit

- **Description** The maximum Cartesian velocity of the robot, in mm/s.

```
from machinelogic import Machine

machine = Machine()
my_robot = machine.get_robot("Robot")
print(my_robot.configuration.cartesian_velocity_limit)
```

joint_velocity_limit

- **Description** The robot joints' maximum angular velocity, in deg/s.

```
from machinelogic import Machine

machine = Machine()
my_robot = machine.get_robot("Robot")
print(my_robot.configuration.joint_velocity_limit)
```

name

- **Description** The friendly name of the robot.

```
from machinelogic import Machine

machine = Machine()
my_robot = machine.get_robot("Robot")
print(my_robot.configuration.name)
```

robot_type

- **Description** The robot's type.

```
from machinelogic import Machine

machine = Machine()
my_robot = machine.get_robot("Robot")
print(my_robot.configuration.robot_type)
```

uuid

- **Description** The robot's ID.

```
from machinelogic import Machine

machine = Machine()
my_robot = machine.get_robot("Robot")
print(my_robot.configuration.uuid)
```

RobotOperationalState

The robot's operational state.

Possible values:

- `OFFLINE`
- `NON_OPERATIONAL`
- `FREEDRIVE`
- `NORMAL`
- `UNKNOWN`
- `NEED_MANUAL_INTERVENTION`

RobotSafetyState

The robot's safety state.

Possible values:

- NORMAL
- EMERGENCY_STOP
- REDUCED_SPEED
- SAFEGUARD_STOP
- UNKNOWN

SequenceBuilder

A builder for a sequence of moves.

append_movej

- **Description** Append a movej to the sequence.
  - **Parameters**
    - **target**
      - **Description** The target joint angles, in degrees.
      - **Type** JointAnglesDegrees
    - **velocity**
      - **Description** The velocity of the move, in degrees per second.
      - **Type** DegreesPerSecond
      - **Default** 10.0
    - **acceleration**
      - **Description** The acceleration of the move, in degrees per second squared.
      - **Type** DegreesPerSecondSquared
      - **Default** 10.0
    - **blend_radius**
      - **Description** The blend radius of the move, in millimeters.
      - **Type** Millimeters
      - **Default** 0.0
  - **Returns**
    - **Description** The builder.
    - **Type** SequenceBuilder

append_movel

- **Description** Append a movel to the sequence.
  - **Parameters**
    - **target**
      - **Description** The target pose.
      - **Type** CartesianPose
    - **velocity**
      - **Description** The velocity of the move, in millimeters per second.
      - **Type** MillimetersPerSecond
      - **Default** 100.0
    - **acceleration**
      - **Description** The acceleration of the move, in millimeters per second squared.
      - **Type** MillimetersPerSecondSquared
      - **Default** 100.0
    - **blend_radius**
      - **Description** The blend radius of the move, in millimeters.
      - **Type** Millimeters
      - **Default** 0.0
    - **reference_frame**
      - **Description** The reference frame for the target pose.

        - **Type** CartesianPose
        - **Default** None
    - **Returns**
        - **Description** The builder.
        - **Type** SequenceBuilder

DigitalInput

A software representation of an DigitalInput. It is not recommended that you construct this object yourself. Rather, you should query it from a Machine instance.

configuration

- **Description** DigitalInputConfiguration: The configuration of the DigitalInput.

state

- **Description** DigitalInputState: The state of the DigitalInput.

```
from machinelogic import Machine

machine = Machine()

my_input = machine.get_input("Input")

if my_input.state.value:
    print(f"{my_input.configuration.name} is HIGH")
else:
    print(f"{my_input.configuration.name} is LOW")
```

DigitalInputState

Representation of the current state of an DigitalInput/DigitalOutput instance.

value

- **Description** bool: The current value of the IO pin. True means high, while False means low. This is different from active/inactive, which depends on the active_high configuration.

DigitalInputConfiguration

Representation of the configuration of an DigitalInput/DigitalOutput. This configuration is established by the configuration page in MachineLogic.

active_high

- **Description** bool: The value that needs to be set to consider the DigitalInput/DigitalOutput as active.

device

- **Description** int: The device number of the DigitalInput/DigitalOutput.

ip_address

- **Description** str: The ip address of the DigitalInput/DigitalOutput.

name

- **Description** str: The name of the DigitalInput/DigitalOutput.

port

- **Description** int: The port number of the DigitalInput/DigitalOutput.

uuid

- **Description** str: The unique ID of the DigitalInput/DigitalOutput.

DigitalOutput

A software representation of an Output. It is not recommended that you construct this object yourself. Rather, you should query it from a Machine instance.

configuration

- **Description** OutputConfiguration: The configuration of the Output.

write

- **Description** Writes the value into the Output, with True being high and False being low.
  - **Parameters**
    - **value**
      - **Description** The value to write to the Output.

- **Type** bool

- **Raises**
  - **Type** DigitalOutputException
    - **Description** If we fail to write the value to the Output.

```
from machinelogic import Machine, MachineException, DigitalOutputException

machine = Machine()
my_output = machine.get_output("Output")

my_output.write(True)  # Write "true" to the Output
my_output.write(False)  # Write "false" to the Output
```

DigitalOutputConfiguration

Representation of the configuration of an DigitalInput/DigitalOutput. This configuration is established by the configuration page in MachineLogic.

active_high

- **Description** bool: The value that needs to be set to consider the DigitalInput/DigitalOutput as active.

device

- **Description** int: The device number of the DigitalInput/DigitalOutput.

ip_address

- **Description** str: The ip address of the DigitalInput/DigitalOutput.

name

- **Description** str: The name of the DigitalInput/DigitalOutput.

port

- **Description** int: The port number of the DigitalInput/DigitalOutput.

uuid

- **Description** str: The unique ID of the DigitalInput/DigitalOutput.

Pneumatic

A software representation of a Pneumatic. It is not recommended that you construct this object yourself. Rather, you should query it from a Machine instance:

E.g.:

```
machine = Machine()
my_pneumatic = machine.get_pneumatic("Pneumatic")
```

In this example, “Pneumatic” is the friendly name assigned to a Pneumatic in the MachineLogic configuration page.

configuration

- **Description** PneumaticConfiguration: The configuration of the actuator.

idle_async

- **Description** Idles the Pneumatic.
  - **Raises**
    - **Type** PneumaticException
      - **Description** If the idle was unsuccessful.

pull_async

- **Description** Pulls the Pneumatic.
  - **Raises**
    - **Type** PneumaticException
      - **Description** If the pull was unsuccessful.

push_async

- **Description** Pushes the Pneumatic.
  - **Raises**
    - **Type** PneumaticException
      - **Description** If the push was unsuccessful.

state

- **Description** PneumaticState: The state of the actuator.

PneumaticConfiguration

Representation of a Pneumatic configuration.

device

- **Description** int: The device of the axis.

input_pin_pull

- **Description** Optional[int]: The optional pull in pin.

input_pin_push

- **Description** Optional[int]: The optional push in pin.

ip_address

- **Description** str: The IP address of the axis.

name

- **Description** str: The name of the Pneumatic.

output_pin_pull

- **Description** int: The pull out pin of the axis.

output_pin_push

- **Description** int: The push out pin of the axis.

uuid

- **Description** str: The ID of the Pneumatic.

ACMotor

A software representation of an AC Motor. It is not recommended that you construct this object yourself. Rather, you should query it from a Machine instance:

E.g.:

```
machine = Machine()
my_ac_motor = machine.get_ac_motor("AC Motor")
```

In this example, “AC Motor” is the friendly name assigned to an AC Motor in the MachineLogic configuration page.

configuration

- **Description** ACMotorConfiguration: The configuration of the ACMotor.

move_forward

- **Description** Begins moving the AC Motor forward.
  - **Raises**
    - **Type** ACMotorException
      - **Description** If the move was unsuccessful.

```
from time import sleep
from machinelogic import Machine

machine = Machine()
my_ac_motor = machine.get_ac_motor("AC Motor")

my_ac_motor.move_forward()
sleep(10)
my_ac_motor.stop()
```

move_reverse

- **Description** Begins moving the AC Motor in reverse.
  - **Raises**
    - **Type** ACMotorException
      - **Description** If the move was unsuccessful.

```
import time
from machinelogic import Machine

machine = Machine()
my_ac_motor = machine.get_ac_motor("AC Motor")

# Move the AC motor in reverse
my_ac_motor.move_reverse()

# The AC motor will stop moving if the program terminates
time.sleep(10)
```

stop

- **Description** Stops the movement of the AC Motor.
  - **Raises**
    - **Type** ACMotorException
      - **Description** If the stop was unsuccessful.

```
import time
from machinelogic import Machine

machine = Machine()

my_ac_motor = machine.get_ac_motor("AC Motor")

# Move the AC Motor forwards
my_ac_motor.move_forward()

# Do something here
time.sleep(10)

my_ac_motor.stop()
```

ACMotorConfiguration

Representation of a ACMotor configuration.

device

- **Description** int: The device of the axis.

ip_address

- **Description** str: The IP address of the axis.

name

- **Description** str: The name of the Pneumatic.

output_pin_direction

- **Description** int: The push out pin of the axis.

output_pin_move

- **Description** int: The pull out pin of the axis.

uuid

- **Description** str: The ID of the Pneumatic.

BagGripper

A software representation of a Bag Gripper. It is not recommended that you construct this object yourself. Rather, you should query it from a Machine instance:

E.g.:

```
machine = Machine()
my_bag_gripper = machine.get_bag_gripper("Bag Gripper")
```

In this example, “Bag Gripper” is the friendly name assigned to a Bag Gripper in the MachineLogic configuration page.

close_async

- **Description** Closes the Bag Gripper.
  - **Raises**
    - **Type** BagGripperException
      - **Description** If the Bag Gripper fails to close.

```
import time
from machinelogic import Machine

machine = Machine()
my_bag_gripper = machine.get_bag_gripper("Bag Gripper")

# Open the Bag Gripper
my_bag_gripper.open_async()

# You can do something while the Bag Gripper is open
time.sleep(10)

# Close the Bag Gripper
my_bag_gripper.close_async()
```

configuration

- **Description** BagGripperConfiguration: The configuration of the actuator.

open_async

- **Description** Opens the Bag Gripper.
  - **Raises**
    - **Type** BagGripperException
      - **Description** If the Bag Gripper fails to open.

```
import time
from machinelogic import Machine

machine = Machine()
my_bag_gripper = machine.get_bag_gripper("Bag Gripper")

# Open the Bag Gripper
my_bag_gripper.open_async()

# You can do something while the Bag Gripper is open
time.sleep(10)

# Close the Bag Gripper
my_bag_gripper.close_async()
```

state

- **Description** BagGripperState: The state of the actuator.

BagGripperConfiguration

Representation of a Bag gripper configuration.

device

- **Description** int: The device of the Bag gripper.

input_pin_close

- **Description** int: The close in pin of the Bag gripper.

input_pin_open

- **Description** int: The open in pin of the Bag gripper.

ip_address

- **Description** str: The IP address of the Bag gripper.

name

- **Description** str: The name of the Bag gripper.

output_pin_close

- **Description** int: The close out pin of the Bag gripper.

output_pin_open

- **Description** int: The open out pin of the Bag gripper.

uuid

- **Description** str: The ID of the Bag gripper.

PathFollower

Path Follower: A Path Follower Object is a group of Actuators, Digital Inputs and Digital Outputs that enable execution of smooth predefined paths. These paths are defined with G-Code instructions. See Vention's G-code interface documentation for a list of supported commands: https://vention.io/resources/guides/path-following-interface-391#parser-configuration

add_tool

- **Description** Add a tool to be referenced by the M(3-5) $ commands
  - **Parameters**
    - **tool_id**
      - **Description** Unique integer defining tool id.
      - **Type** int
    - **m3_output**
      - **Description** Output to map to the M3 Gcode command
      - **Type** IDigitalOutput
    - **m4_output**
      - **Description** Optional, Output to map to the M4 Gcode command
      - **Type** Union[IDigitalOutput, None]
  - **Raises**
    - **Type** PathFollowerException
      - **Description** If the tool was not properly added.

```
from machinelogic import Machine, PathFollower

machine = Machine()
x_axis = machine.get_actuator("X_Actuator")
y_axis = machine.get_actuator("Y_Actuator")
m3_output = machine.get_output("M3_Output")
m4_output = machine.get_output("M4_Output")

GCODE = """
G90   ; Absolute position mode
G0 X60 Y110 F1200 ; Rapid move at 1200 mm/minute
M3 $1 ; Start tool 1 in clockwise direction
G1 X110 Y110 F1000 ; Travel move at 1000 mm/minute
M4 $1 ; counter clockwise now
G0 X50 Y50
M5 $1 ; Stop tool 1
G0 X1 Y1
"""

# Create PathFollower instance
path_follower = PathFollower(x_axis, y_axis)

# Associate a digital output with a GCode tool number
path_follower.add_tool(1, m3_output, m4_output)  # clockwise  # counterclockwise

path_follower.start_path(GCODE)
```

start_path

- **Description** Start the path, returns when path is complete
  - **Parameters**
    - **gcode**
      - **Description** Gcode path
      - **Type** str
  - **Raises**
    - **Type** PathFollowerException
      - **Description** If failed to run start_path

```
from machinelogic import Machine, PathFollower

machine = Machine()

# must be defined in ControlCenter
x_axis = machine.get_actuator("X_Actuator")
y_axis = machine.get_actuator("Y_Actuator")

GCODE = """
(Operational mode commands)
G90   ; Absolute position mode
G90.1 ; Absolute arc centre
G21   ; Use millimeter units
G17   ; XY plane arcs
G64 P0.5 ; Blend move mode, 0.5 mm tolerance
(Movement and output commands)
G0 X60 Y110 F1200 ; Rapid move at 1200 mm/minute
G1 X110 Y110 F1000 ; Travel move at 1000 mm/minute
G2 X110 Y10 I100 J60 ; Clockwise arc
G1 X60 Y10
G2 X60 Y110 I60 J60
G4 P1.0 ; Dwell for 1 second
G0 X1 Y1
"""

# Create PathFollower instance
path_follower = PathFollower(x_axis, y_axis)

# Run your Gcode
path_follower.start_path(GCODE)
```

start_path_async

- **Description** Start the path, nonblocking, returns immediately
  - **Parameters**
    - **gcode**
      - **Description** Gcode path
      - **Type** str
  - **Raises**
    - **Type** PathFollowerException
      - **Description** If failed to run start_path_async

```
from machinelogic import Machine, PathFollower
import time

machine = Machine()

# must be defined in ControlCenter
x_axis = machine.get_actuator("X_Actuator")
y_axis = machine.get_actuator("Y_Actuator")

GCODE = """
(Operational mode commands)
G90   ; Absolute position mode
G90.1 ; Absolute arc centre
G21   ; Use millimeter units
G17   ; XY plane arcs
G64 P0.5 ; Blend move mode, 0.5 mm tolerance
(Movement and output commands)
G0 X60 Y110 F1200 ; Rapid move at 1200 mm/minute
G1 X110 Y110 F1000 ; Travel move at 1000 mm/minute
G2 X110 Y10 I100 J60 ; Clockwise arc
G1 X60 Y10
G2 X60 Y110 I60 J60
G4 P1.0 ; Dwell for 1 second
G0 X1 Y1
"""

path_follower = PathFollower(x_axis, y_axis)

# Run your Gcode
path_follower.start_path_async(GCODE)

PATH_IN_PROGRESS = True
while PATH_IN_PROGRESS:
    time.sleep(0.5)
    path_state = path_follower.state
    PATH_IN_PROGRESS = path_state.running
    print(
        {
            "running": path_state.running,
            "line_number": path_state.line_number,
            "current_command": path_state.current_command,
            "error": path_state.error,
            "speed": path_state.speed,
            "acceleration": path_state.acceleration,
        }
    )
```

state

- **Description** Current state of the path follower

stop_path

- **Description** Abruptly stop the path following procedure. Affects all actuators in the PathFollower instance
  - **Raises**
    - **Type** PathFollowerException
      - **Description** If failed to stop path

wait_for_path_completion

- **Description** Wait for the path to complete

```
from machinelogic import Machine, PathFollower

machine = Machine()

# must be defined in ControlCenter
x_axis = machine.get_actuator("X_Actuator")
y_axis = machine.get_actuator("Y_Actuator")

GCODE = """
(Operational mode commands)
G90   ; Absolute position mode
G90.1 ; Absolute arc centre
G21   ; Use millimeter units
G17   ; XY plane arcs
G64 P0.5 ; Blend move mode, 0.5 mm tolerance
(Movement and output commands)
G0 X60 Y110 F1200 ; Rapid move at 1200 mm/minute
G1 X110 Y110 F1000 ; Travel move at 1000 mm/minute
G2 X110 Y10 I100 J60 ; Clockwise arc
G1 X60 Y10
G2 X60 Y110 I60 J60
G4 P1.0 ; Dwell for 1 second
G0 X1 Y1
"""

# Create PathFollower instance
path_follower = PathFollower(x_axis, y_axis)

# Run your Gcode asynchronously
path_follower.start_path_async(GCODE)

# Waits for path completion before continuing with program
path_follower.wait_for_path_completion()
print("Path Complete")
```

PathFollowerState

PathFollower State

acceleration

- **Description** float: The current tool acceleration in millimeters/second^2

current_command

- **Description** Union[str, None]: In-progress command of gcode script.

error

- **Description** Union[str, None]: A description of errors encountered during path execution.

line_number

- **Description** int: Current line number in gcode script.

running

- **Description** bool: True if path following in progress.

speed

- **Description** float: The current tool speed in millimeters/second

Scene

A software representation of the scene containing assets that describe and define reference frames and targets for robots.

Only a single instance of this object should exist in your program.

get_calibration_frame

- **Description** Gets a calibration frame from scene assets by name
  - **Parameters**
    - **name**
      - **Description** Friendly name of the calibration frame asset
      - **Type** str
  - **Returns**
    - **Description** The found calibration frame
    - **Type** ICalibrationFrame
  - **Raises**
    - **Type** SceneException
      - **Description** If the scene asset is not found

```
from machinelogic import Machine

machine = Machine()
scene = machine.get_scene()

# Assuming we have a calibration frame defined
# in Scene Assets called "calibration_frame_1"
calibration_frame = scene.get_calibration_frame("calibration_frame_1")

default_value = calibration_frame.get_default_value()

print(default_value)
```

CalibrationFrame

A calibration frame, as defined in the scene assets pane, is represented in software. It is measured in millimeters and degrees, with angles given as extrinsic Euler angles in XYZ order.

get_calibrated_value

- **Description** Gets the calibration frame's calibrated values.
  - **Returns**
    - **Description** The calibrated value of the calibration frame
      in mm and degrees, where the angles are extrinsic Euler angles in XYZ order.
    - **Type** CartesianPose
  - **Raises**
    - **Type** SceneException
      - **Description** If failed to get the calibrated value

```
from machinelogic import Machine

machine = Machine()
scene = machine.get_scene()

# Assuming we have a calibration frame defined
# in Scene Assets called "calibration_frame_1"
calibration_frame = scene.get_calibration_frame("calibration_frame_1")

calibrated_value = calibration_frame.get_calibrated_value()

print(calibrated_value)
```

get_default_value

- **Description** Gets the calibration frame's default values.
  - **Returns**
    - **Description** The nominal value of the calibration frame
      in mm and degrees, where the angles are extrinsic Euler angles in XYZ order.
    - **Type** CartesianPose

- **Raises**
  - **Type** SceneException
    - **Description** If failed to get the default value

```
from machinelogic import Machine

machine = Machine()
scene = machine.get_scene()

# Assuming we have a calibration frame defined
# in Scene Assets called "calibration_frame_1"
calibration_frame = scene.get_calibration_frame("calibration_frame_1")

default_value = calibration_frame.get_default_value()

print(default_value)
```

set_calibrated_value

- **Description** Sets the calibration frame's calibrated values.
  - **Parameters**
    - **frame**
      - **Description** The calibrated values of the Calibration Frame in mm and degrees, where the angles are extrinsic Euler angles in XYZ order.
      - **Type** CartesionPose
  - **Raises**
    - **Type** SceneException
      - **Description** If failed to set the calibrated value

```
from machinelogic import Machine

machine = Machine()
scene = machine.get_scene()

# Assuming we have a calibration frame defined
# in Scene Assets called "calibration_frame_1"
calibration_frame = scene.get_calibration_frame("calibration_frame_1")

# CartesianPose in mm and degrees, where the angles are
# extrinsic Euler angles in XYZ order.
calibrated_cartesian_pose = [100, 100, 50, 90, 90, 0]

calibration_frame.set_calibrated_value(calibrated_cartesian_pose)
```

Exceptions

ActuatorException

An exception thrown by an Actuator

Args:

VentionException (VentionException): Super class

args

- **Description**

with_traceback

- **Description** Exception.with_traceback(tb) – set self.**traceback** to tb and return self.

MachineException

An exception thrown by the Machine

Args:

Exception (VentionException): Super class

args

- **Description**

with_traceback

- **Description** Exception.with_traceback(tb) – set self.**traceback** to tb and return self.

RobotException

An exception thrown by a Robot

Args:

VentionException (VentionException): Super class

args

- **Description**

with_traceback

- **Description** Exception.with_traceback(tb) – set self.**traceback** to tb and return self.

MachineMotionException

An exeption thrown by a MachineMotion

Args:

```
    VentionException (VentionException): Super class
```

args

- **Description**

with_traceback

- **Description** Exception.with_traceback(tb) – set self.**traceback** to tb and return self.

DigitalInputException

An exception thrown by an INput

Args:

```
    VentionException (VentionException): Super class
```

args

- **Description**

with_traceback

- **Description** Exception.with_traceback(tb) – set self.**traceback** to tb and return self.

DigitalOutputException

An exception thrown by an Output

Args:

VentionException (VentionException): Super class

args

- **Description**

with_traceback

- **Description** Exception.with_traceback(tb) – set self.**traceback** to tb and return self.

ActuatorGroupException

An exception thrown by an ActuatorGroup

Args:

VentionException (VentionException): Super class

args

- **Description**

with_traceback

- **Description** Exception.with_traceback(tb) – set self.**traceback** to tb and return self.

EstopException

An exception thrown by the Machine

Args:

Exception (VentionException): Super class

args

- **Description**

with_traceback

- **Description** Exception.with_traceback(tb) – set self.**traceback** to tb and return self.

PathFollowingException

An exception thrown by a path following operation

Args:

```
    VentionException(__type__): Super class
```

args

- **Description**

with_traceback

- **Description** Exception.with_traceback(tb) – set self.**traceback** to tb and return self.