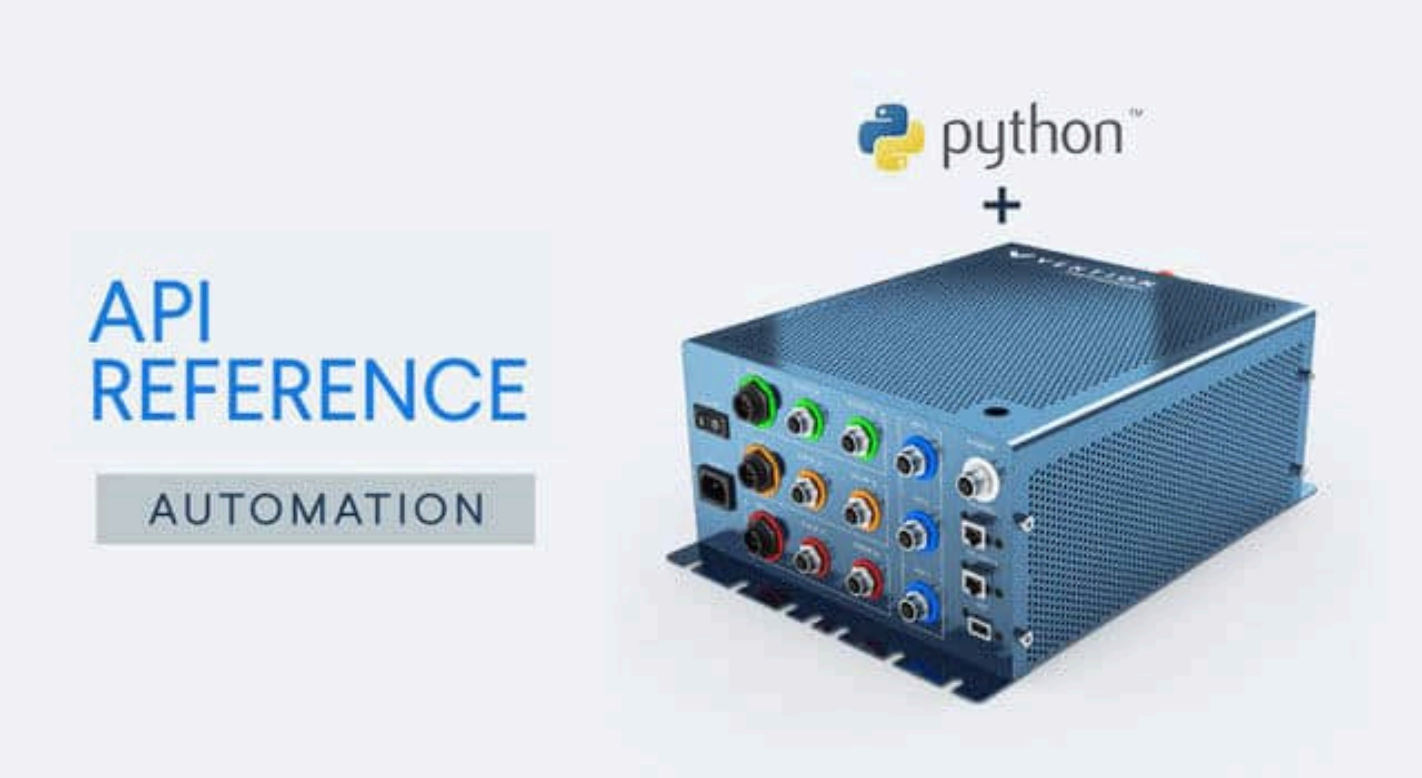
USER MANUAL

Updated: July 16th, 2021

Application Programming Interface: Python v2.2

Version 2.2



Overview

This is the developers reference manual for version 2.2 of MachineMotion's Python API. This API is compatible with Python version 2.7.

If this is your first time using the MachineMotion controller, please follow the Python programming manual here. (https://www.vention.io/technical-documents/programming-manual-python-55)

Version 2.2 of the API is compatible with MachineMotion controller software version 1.2.11. If your controller software is 1.12 or later, please refer to the latest version of the python API.

To get your controller software version, refer to the user Manual here (https://vention.io/resources/guides/machine-motion-user-manual-v1e-82#network-configuration).

Download

The Python API comes pre-installed on your MachineMotion controller. To update the API verison or download it to an external computer, follow this link (https://github.com/VentionCo/mm-python-api/tree/release/v2.2) to version 2.2 of the Python API on Vention's Github.

API Reference

configAxis (*axis, uStep, mechGain*)

Description

Configures motion parameters for a single axis.

Parameters

`axis` (Required) `Number` - The axis to configure.
`uStep` (Required) `Number` - The microstep setting of the axis.
`mechGain` (Required) `Number` - The distance moved by the actuator for every full rotation of the stepper motor, in mm/revolution.

Return Value

None

Source Code

```
    def configAxis(self, axis, uStep, mechGain):
        self._restrictInputValue("axis", axis,  AXIS_NUMBER)
        self._restrictInputValue("uStep", uStep, MICRO_STEPS)

        uStep    = float(uStep)
        mechGain = float(mechGain)

        if(axis == 1):
            self.myAxis1_steps_mm = 200 * uStep / mechGain
            self.myGCode.__emit__("M92 " + self.myGCode.__getTrueAxis__(axis) + str(self.myA
        elif(axis == 2):
            self.myAxis2_steps_mm = 200 * uStep / mechGain
            self.myGCode.__emit__("M92 " + self.myGCode.__getTrueAxis__(axis) + str(self.myA
        elif(axis == 3):
            self.myAxis3_steps_mm = 200 * uStep / mechGain
            self.myGCode.__emit__("M92 " + self.myGCode.__getTrueAxis__(axis) + str(self.myA
```

Example Code

Copy Code

```
import sys
sys.path.append("..")
from _MachineMotion import *

mm = MachineMotion(DEFAULT_IP_ADDRESS.usb_windows)

# Configure the axis number 1, 8 uSteps and 150 mm / turn for a timing belt
axis = AXIS_NUMBER.DRIVE1
uStep = MICRO_STEPS.ustep_8
mechGain = MECH_GAIN.timing_belt_150mm_turn
mm.configAxis(axis, uStep, mechGain)
print("Axis " + str(axis) + " configured with " + str(uStep) + " microstepping and " + str(me
```

NOTE:
The uStep setting is hardcoded into the machinemotion controller through a DIP switch and is by default set to 8. The value here must match the value on the DIP Switch.

configAxisDirection (*axis, direction*)

Description

Configures a single axis to operate in either clockwise (normal) or counterclockwise (reverse) mode. Refer to the Automation System Diagram for the correct axis setting.

Parameters

`axis` (Required) `Number` - The specified axis.
`direction` (Required) `String` - A string of value either either 'Normal' or 'Reverse'. 'Normal' direction means the axis will home towards end stop sensor A and reverse will make the axis home towards end stop B.

Return Value

None

Source Code

```
    def configAxisDirection(self, axis, direction):

        self._restrictInputValue("axis", axis, AXIS_NUMBER)
        self._restrictInputValue("direction", direction, AXIS_DIRECTION)

        if(axis == 1):
            self.myAxis1_direction = direction
            if(direction == AXIS_DIRECTION.normal):
                self.myGCode.__emit__("M92 " + self.myGCode.__getTrueAxis__(axis) + str(self.
            elif (direction == AXIS_DIRECTION.reverse):
                self.myGCode.__emit__("M92 " + self.myGCode.__getTrueAxis__(axis) + "-" + str
        elif(axis == 2):
            if(direction == AXIS_DIRECTION.normal):
                self.myGCode.__emit__("M92 " + self.myGCode.__getTrueAxis__(axis) + str(self.
            elif (direction == AXIS_DIRECTION.reverse):
                self.myGCode.__emit__("M92 " + self.myGCode.__getTrueAxis__(axis) + "-" + str
        elif(axis == 3):
            if(direction == AXIS_DIRECTION.normal):
                self.myGCode.__emit__("M92 " + self.myGCode.__getTrueAxis__(axis) + str(self.
            elif (direction == AXIS_DIRECTION.reverse):
                self.myGCode.__emit__("M92 " + self.myGCode.__getTrueAxis__(axis) + "-" + str
```

Example Code

Copy Code

```
import sys
sys.path.append("..")
from _MachineMotion import *

mm = MachineMotion(DEFAULT_IP_ADDRESS.usb_windows)

axis = AXIS_NUMBER.DRIVE1
mm.configAxis(axis, MICRO_STEPS.ustep_8, MECH_GAIN.timing_belt_150mm_turn)

homesTowards={
    AXIS_DIRECTION.positive: "sensor " + str(axis) + "A",
    AXIS_DIRECTION.negative: "sensor " + str(axis) + "B"
}

direction = AXIS_DIRECTION.positive
print("Axis " + str(axis) + " is set to " + direction + " mode. It will now home towards " +
mm.configAxisDirection(axis, direction)
mm.emitHome(axis)

mm.waitForMotionCompletion()

direction = AXIS_DIRECTION.negative
print("Axis " + str(axis) + " is set to " + direction + " mode. It will now home towards " +
mm.configAxisDirection(axis, direction)
mm.emitHome(axis)

mm.waitForMotionCompletion()
```

NOTE:
For more details on how to properly set the axis direction, please see here (https://vention-demo.herokuapp.com/technical-documents/machine-motion-user-manual-123#actuator-hardware-configuration)

configHomingSpeed (*axes, speeds, units*)

Description

Sets homing speed for all 3 axes.

Parameters

axes (Required) `List` - A list of the axes to configure. ex - [1,2,3]
speeds (Required) `List` - A list of homing speeds to set for each axis. ex - [50, 50, 100]
units (Optional. Default = UNITS_SPEED.mm_per_sec) `String` - Units for speed. Can be switched to UNITS_SPEED.mm_per_min

Return Value

None

Source Code

```
    def configHomingSpeed(self, axes, speeds, units = UNITS_SPEED.mm_per_sec):
        try:
            axes = list(axes)
            speeds = list(speeds)
        except TypeError:
            axes = [axes]
            speeds = [speeds]

        if len(axes) != len(speeds):
            class InputsError(Exception):
                pass
            raise InputsError("axes and speeds must be of same length")

        gCodeCommand = "V2 "
        for idx, axis in enumerate(axes):

            if units == UNITS_SPEED.mm_per_sec:
                speed_mm_per_min = speeds[idx] * 60
            elif units == UNITS_SPEED.mm_per_min:
                speed_mm_per_min = speeds[idx]



            if speed_mm_per_min < HARDWARE_MIN_HOMING_FEEDRATE:
                raise self.HomingSpeedOutOfBounds("Your desired homing speed of " + str(speed
            if speed_mm_per_min > HARDWARE_MAX_HOMING_FEEDRATE:
                raise self.HomingSpeedOutOfBounds("Your desired homing speed of " + str(speed

            gCodeCommand = gCodeCommand + self.myGCode.__getTrueAxis__(axis) + str(speed_mm_p

        while self.isReady() == False: pass
        gCodeCommand = gCodeCommand.strip()
        self.emitgCode(gCodeCommand)
```

Example Code

Copy Code

```
import sys
sys.path.append("..")
from _MachineMotion import *

mm = MachineMotion(DEFAULT_IP_ADDRESS.usb_windows)

axes = [1,2,3]
axis =3                                    #The axis that you'd like to move
homingSpeeds = [50,50,50]                        #The homing speeds to set for each axis

mm.emitSpeed(100)
mm.emitAcceleration(50)
mm.configAxis(axis, MICRO_STEPS.ustep_8, MECH_GAIN.timing_belt_150mm_turn)

print("Moving to position = 100")
mm.emitAbsoluteMove(axis, 100)
mm.waitForMotionCompletion()

#Sets minimum and maximum allowable homing speeds for each axis
minHomingSpeeds = [20, 20, 20]
maxHomingSpeeds = [400, 400, 400]
mm.configMinMaxHomingSpeed(axes,minHomingSpeeds, maxHomingSpeeds, UNITS_SPEED.mm_per_sec)

#Sets homing speeds for all three axes. The selected homing speed must be within the range se
mm.configHomingSpeed(axes, homingSpeeds)
mm.waitForMotionCompletion()

#Homes the axis at the newly configured homing speed.
mm.emitHome(axis)
mm.waitForMotionCompletion()
```

NOTE:
Once set, the homing speed will apply to all programs, including MachineLogic applications.

configMachineMotionIp (*mode, machineIp, machineNetmask, machineGateway*)

Description

Set up the required network information for the Machine Motion controller. The router can be configured in either DHCP mode or static mode.

Parameters

mode (Required) `Constant` - Sets Network Mode to either DHCP or static addressing. Either `NETWORK_MODE.static` or `NETWORK_MODE.dhcp`
machineIp (Required) `String` - The static IP Address given to the controller. (Required if mode = `NETWORK_MODE.static`)
machineNetmask (Required) `String` - The netmask IP Address given to the controller. (Required if mode = `NETWORK_MODE.static`)
machineGateway (Required) `String` - The gateway IP Address given to the controller. (Required if mode = `NETWORK_MODE.static`)

Return Value

None

Source Code

```
    def configMachineMotionIp(self, mode, machineIp="", machineNetmask="", machineGateway="")

        if(mode == NETWORK_MODE.static):
            if '' in [machineIp, machineNetmask, machineGateway]:
               print("NETWORK ERROR: machineIp, machineNetmask and machineGateway cannot be
               quit()

        # Create a new object and augment it with the key value.
        self.myConfiguration["mode"] = mode
        self.myConfiguration["machineIp"] = machineIp
        self.myConfiguration["machineNetmask"] = machineNetmask
        self.myConfiguration["machineGateway"] = machineGateway

        self.mySocket.emit('configIp', json.dumps(self.myConfiguration))
        time.sleep(1)
```

Example Code

Copy Code

```
import sys
sys.path.append("..")
from MachineMotion import *

#Initialize MachineMotion with default IP address
mm = MachineMotion(DEFAULT_IP_ADDRESS.usb_windows)
print("Application Message: MachineMotion Controller Connected")

#Configure machine motion with a static IP address
mode = NETWORK_MODE.static
machineIp = "192.168.0.2"
machineNetmask="255.255.255.0"
machineGateway = "192.168.0.1"
mm.configMachineMotionIp(mode, machineIp, machineNetmask, machineGateway)
print("MachineMotion configured in static mode with an ip of " + str(machineIp) +".")


#Configure MachineMotion IP to use Dynamic Host Configuration Protocol (DHCP)
mm.configMachineMotionIp(NETWORK_MODE.dhcp)
print("MachineMotion configured in DHCP mode.")
```

configMinMaxHomingSpeed(*axes, minspeeds, maxspeeds*)

Description

Sets the minimum and maximum homing speeds for each axis.

Parameters

axes (Required) `List` - a list of the axes that require minimum and maximum homing speeds.
minspeeds (Required) `List` - the minimum speeds for each axis.
maxspeeds (Required) `List` - the maximum speeds for each axis, in the same order as the axes parameter

Return Value

None

Source Code

```
    def configMinMaxHomingSpeed(self, axes, minspeeds, maxspeeds, units = UNITS_SPEED.mm_per_
        gCodeCommand = "V1 "
        for idx, axis in enumerate(axes):

            if units == UNITS_SPEED.mm_per_sec:
                min_speed_mm_per_min = minspeeds[idx] * 60
                max_speed_mm_per_min = maxspeeds[idx] * 60
            elif units == UNITS_SPEED.mm_per_min:
                min_speed_mm_per_min = minspeeds[idx]
                max_speed_mm_per_min = maxspeeds[idx]

            if min_speed_mm_per_min < HARDWARE_MIN_HOMING_FEEDRATE:
                raise self.HomingSpeedOutOfBounds("Your desired homing speed of " + str(min_s
            if max_speed_mm_per_min > HARDWARE_MAX_HOMING_FEEDRATE:
                raise self.HomingSpeedOutOfBounds("Your desired homing speed of " + str(max_s

            gCodeCommand = gCodeCommand + self.myGCode.__getTrueAxis__(axis) + str(min_speed_

        while self.isReady() == False: pass
        gCodeCommand = gCodeCommand.strip()
        self.emitgCode(gCodeCommand)
```

Example Code

Copy Code

```
import sys
sys.path.append("..")
from _MachineMotion import *

mm = MachineMotion(DEFAULT_IP_ADDRESS.usb_windows)

axes = [1,2,3]
axis =3                                    #The axis that you'd like to move
homingSpeeds = [50,50,50]                        #The homing speeds to set for each axis

mm.emitSpeed(100)
mm.emitAcceleration(50)
mm.configAxis(axis, MICRO_STEPS.ustep_8, MECH_GAIN.timing_belt_150mm_turn)

print("Moving to position = 100")
mm.emitAbsoluteMove(axis, 100)
mm.waitForMotionCompletion()

#Sets minimum and maximum allowable homing speeds for each axis
minHomingSpeeds = [20, 20, 20]
maxHomingSpeeds = [400, 400, 400]
mm.configMinMaxHomingSpeed(axes,minHomingSpeeds, maxHomingSpeeds, UNITS_SPEED.mm_per_sec)

#Sets homing speeds for all three axes. The selected homing speed must be within the range se
mm.configHomingSpeed(axes, homingSpeeds)
mm.waitForMotionCompletion()

#Homes the axis at the newly configured homing speed.
mm.emitHome(axis)
mm.waitForMotionCompletion()
```

NOTE:
This function can be used to set safe limits on homing speed. Because homing speed is configured only through software aPI, this safeguards against developers accidently modifying homing speed to unsafe levels.

detectIOModules ( )

Description

Returns a dictionary containing all detected IO Modules.

Return Value

`Dictionary` Dictionary with keys of format "Digital IO Network Id [id]" and values [id] where [id] is the network IDs of all connected digital IO modules.

Source Code

```
    def detectIOModules(self):
        class NoIOModulesFound(Exception):
            pass

        foundIOModules = {}
        numIOModules = 0

        for ioDeviceID in range(0,3):
            if self.isIoExpanderAvailable(ioDeviceID):
                foundIOModules["Digital IO Network Id " + str(ioDeviceID)] = ioDeviceID
                numIOModules = numIOModules + 1

        if numIOModules == 0:
            raise NoIOModulesFound("Application Error: No IO Modules found. Please verify the
        else:
            return foundIOModules
```

Example Code

Copy Code

```
# System imports
import sys
# Custom imports
sys.path.append("..")

from _MachineMotion import *

mm = MachineMotion(DEFAULT_IP_ADDRESS.usb_windows)


#Reads and Prints all input values on all connected digital IO Modules
detectedIOModules = mm.detectIOModules()
for IO_Name, IO_NetworkID in detectedIOModules.items():
   readPins = {"Pin 1":0, "Pin 2":1, "Pin 3": 2, "Pin 4":3}
   for readPin in readPins:
       pinValue = mm.digitalRead(IO_NetworkID, readPins[readPin])
       print(readPin + " on " + IO_Name + " has value " + str(pinValue))
```

NOTE:
For more information, please see the digital IO datasheet here (https://www.vention.io/technical-documents/digital-io-module-datasheet-70)

digitalRead (*deviceNetworkId, pin*)

Description

Reads the state of a digital IO modules input pins.

Parameters

`deviceNetworkId` (Required) `Integer` - The IO Modules device network ID. It can be found printed on the product sticker on the back of the digital IO module.
`pin` (Required) `Integer` - The index of the input pin.

Return Value

`Integer` Returns 1 if the input pin is 24V and returns 0 if the input pin is 0V.

Source Code

```
    def digitalRead(self, deviceNetworkId, pin):

        if ( not self.isIoExpanderInputIdValid( deviceNetworkId, pin ) ):
            logging.warning("DEBUG: unexpected digital-output parameters: device= " + str(dev
            return
        if (not hasattr(self, 'digitalInputs')):
            self.digitalInputs = {}
        if (not deviceNetworkId in self.digitalInputs):
            self.digitalInputs[deviceNetworkId] = {}
        if (not pin in self.digitalInputs[deviceNetworkId]):
            self.digitalInputs[deviceNetworkId][pin] = 0
        return self.digitalInputs[deviceNetworkId][pin]
```

Example Code

Copy Code

```
# System imports
import sys
# Custom imports
sys.path.append("..")

from _MachineMotion import *

mm = MachineMotion(DEFAULT_IP_ADDRESS.usb_windows)


#Reads and Prints all input values on all connected digital IO Modules
detectedIOModules = mm.detectIOModules()
for IO_Name, IO_NetworkID in detectedIOModules.items():
   readPins = {"Pin 1":0, "Pin 2":1, "Pin 3": 2, "Pin 4":3}
   for readPin in readPins:
       pinValue = mm.digitalRead(IO_NetworkID, readPins[readPin])
       print(readPin + " on " + IO_Name + " has value " + str(pinValue))
```

NOTE:
The pin labels on the digital IO module (pin 1, pin 2, pin 3, pin 4) correspond in software to (0, 1, 2, 3). Therefore, digitalRead(deviceNetworkId, 2) will read the value on input pin 3.

digitalWrite (*deviceNetworkId, pin, value*)

Description

Sets voltage on specified pin of digital IO output pin to either 24V or 0V.

Parameters

`deviceNetworkId` (Required) `Integer` - The IO Modules device network ID. It can be found printed on the product sticker on the back of the digital IO module.
`pin` (Required) `Integer` - The output pin number to write to.
`value` (Required) `Integer` - Writing 1 will set digial output to 24V, writing 0 will set digital output to 0V.

Return Value

None

Source Code

```
    def digitalWrite(self, deviceNetworkId, pin, value):
        if (self.isIoExpanderOutputIdValid( deviceNetworkId, pin ) == False):
            print ( "DEBUG: unexpected digitalOutput parameters: device= " + str(deviceNetwor
            return
        self.myMqttClient.publish('devices/io-expander/' + str(deviceNetworkId) + '/digital-o
```

Example Code

Copy Code

```
import sys
sys.path.append("..")
from _MachineMotion import *

mm = MachineMotion(DEFAULT_IP_ADDRESS.usb_windows)

# Toggles the output pins on all connected IO Modules
detectedIOModules = mm.detectIOModules()
for IO_Name, IO_NetworkID in detectedIOModules.items():
    writePins ={"Pin 1":0, "Pin 2":1, "Pin 3": 2, "Pin 4":3}
    for writePin in writePins.keys():
        print(writePin + " on " + IO_Name + " is going to flash twice")

        mm.digitalWrite(IO_NetworkID, writePins[writePin], 1)
        time.sleep(1)
        mm.digitalWrite(IO_NetworkID, writePins[writePin], 0)
        time.sleep(1)
        mm.digitalWrite(IO_NetworkID, writePins[writePin], 1)
        time.sleep(1)
        mm.digitalWrite(IO_NetworkID, writePins[writePin], 0)
```

NOTE:
Output pins maximum sourcing current is 75 mA and the maximum sinking current is 100 mA. The pin labels on the digital IO module (pin 1, pin 2, pin 3, pin 4) correspond in software to (0, 1, 2, 3). Therefore, digitalWrite(deviceNetworkId, 2, 1) will set output pin 3 to 24V.

emitAbsoluteMove (*axis, position*)

Description

Moves the specified axis to a desired end location.

Parameters

`axis` (Required) `Number` - The axis which will perform the absolute move command.

`position` (Required) `Number` - The desired end position of the axis movement.

Return Value

None

Source Code

```
    def emitAbsoluteMove(self, axis, position):
        self._restrictInputValue("axis", axis, AXIS_NUMBER)
        global motion_completed

        motion_completed = "false"

        # Set to absolute motion mode
        self.myGCode.__emit__("G90")
        while self.isReady() != "true": pass

        # Transmit move command
        self.myGCode.__emit__("G0 " + self.myGCode.__getTrueAxis__(axis) + str(position))
        while self.isReady() != "true": pass
```

Example Code

Copy Code

```
import sys
sys.path.append("..")
from _MachineMotion import *

# Define a callback to process controller gCode responses (if desired)
def templateCallback(data):
   print ( "Controller gCode responses " + data )

mm = MachineMotion(DEFAULT_IP_ADDRESS.usb_windows, gCodeCallback = templateCallback)

axis = 1                                        #The axis that you'd like to move
speed = 400                                     #The max speed you'd like to move at
acceleration = 500                              #The constant acceleration and decceleration v
position = 100                                  #The absolute position you'd like to move to
mechGain = MECH_GAIN.timing_belt_150mm_turn     #The mechanical gain of the actuator on the ax
mm.configAxis(axis, MICRO_STEPS.ustep_8, mechGain)

# Home Axis before absolute move
mm.emitHome(axis)
print("Axis " + str(axis) + " is going home.")
mm.waitForMotionCompletion()
print("Axis " + str(axis) + " homed.")

# Configure movement speed, acceleration and then move
mm.emitSpeed(speed)
mm.emitAcceleration(acceleration)
mm.emitAbsoluteMove(axis, position)
print("Axis " + str(axis) + " is moving towards position " + str(position) + "mm")
mm.waitForMotionCompletion()
print("Axis " + str(axis) + " is at position " + str(position) + "mm")
```

emitAcceleration (*mm_per_sec_sqr, units*)

Description

Sets the global acceleration for all movement commands on all axes.

Parameters

mm_per_sec_sqr (Required) `Number` - The global acceleration in mm/s^2.

units (Optional. Default = UNITS_ACCEL.mm_per_sec_sqr) `String` - Units for speed. Can be switched to UNITS_ACCEL.mm_per_min_sqr

Return Value

None

Source Code

```
    def emitAcceleration(self, acceleration, units=UNITS_ACCEL.mm_per_sec_sqr):

        self._restrictInputValue("units", units, UNITS_ACCEL)

        if units == UNITS_ACCEL.mm_per_sec_sqr:
            accel_mm_per_sec_sqr = acceleration
        elif units == UNITS_ACCEL.mm_per_min_sqr:
            accel_mm_per_sec_sqr = acceleration/3600

        self.myGCode.__emit__("M204 T" + str(accel_mm_per_sec_sqr))
        while self.isReady() != "true": pass
```

Example Code

Copy Code

```
import sys
sys.path.append("..")
from _MachineMotion import *

# Define a callback to process controller gCode responses (if desired)
def templateCallback(data):
   print ( "Controller gCode responses " + data )

mm = MachineMotion(DEFAULT_IP_ADDRESS.usb_windows, gCodeCallback = templateCallback)

acceleration = 500      # The acceleration [mm/s^2] that all subsequent moves will move at
mm.emitAcceleration(acceleration)
print("Global acceleration set to " + str(acceleration))
```

emitCombinedAxesAbsoluteMove (*axes, positions*)

Description

Moves multiple specified axes to their desired end locations synchronously.

Parameters

axes (Required) `List` - The axes which will perform the move commands. Ex - [1 ,3]
positions (Required) `List` - The desired end position of all axess movement. Ex - [50, 10]

Return Value

None

Source Code

```
    def emitCombinedAxesAbsoluteMove(self, axes, positions):

        try:
            axes = list(axes)
            positions = list(positions)
        except TypeError:
            raise TypeError("Axes and Postions must be either lists or convertible to lists")

        for axis in axes:
            self._restrictInputValue("axis", axis, AXIS_NUMBER)

        global motion_completed

        motion_completed = "false"

        # Set to absolute motion mode
        self.myGCode.__emit__("G90")
        while self.isReady() != "true": pass

        # Transmit move command
        command = "G0 "
        for axis, position in zip(axes, positions):
            command += self.myGCode.__getTrueAxis__(axis) + str(position) + " "
        self.myGCode.__emit__(command)
        while self.isReady() != "true": pass
```

Example Code

Copy Code

```
import sys
sys.path.append("..")
from _MachineMotion import *


#Declare parameters for combined absolute move
speed = 500
acceleration = 500
axesToMove = [1,2,3]
positions = [50, 100,50]
mechGain = MECH_GAIN.timing_belt_150mm_turn

# Define a callback to process controller gCode responses (if desired)
def templateCallback(data):
   print ( "Controller gCode responses " + data )

mm = MachineMotion(DEFAULT_IP_ADDRESS.usb_windows, gCodeCallback = templateCallback)

mm.emitSpeed(speed)
mm.emitAcceleration(acceleration)
for axis in axesToMove:
    mm.configAxis(axis, MICRO_STEPS.ustep_8, mechGain)
print("All Axes Moving Home Sequentially")
mm.emitHomeAll()
mm.waitForMotionCompletion()
print("All Axes homed.")

# Simultaneously moves three axis:
#   Moves axis 1 to absolute position 50
#   Moves axis 2 to absolute position 100
#   Moves axis 3 to absolute position 50
mm.emitCombinedAxesAbsoluteMove(axesToMove, positions)
mm.waitForMotionCompletion()
for index, axis in enumerate(axesToMove):
    print("Axis " + str(axis) + " moved to position " + str(positions[index]))
```

NOTE:
The current speed and acceleration settings are applied to the combined motion of the axes.

emitCombinedAxisRelativeMove (*axes, directions, distances*)

Description

Moves the multiple specified axes the specified distances in the specified directions.

Parameters

axes (Required) `List of Integers` - The axes to move. Ex-[1,3]
directions (Required) `List of Strings` - The direction of travel of each specified axis. Ex - ["positive", "negative"]
distances (Required) `List of Numbers` - The travel distances in mm. Ex - [10, 40]

Return Value

None

Source Code

```
    def emitCombinedAxisRelativeMove(self, axes, directions, distances):

        try:
            axes = list(axes)
            directions = list(directions)
            distances = list(distances)
        except TypeError:
            raise TypeError("Axes, positions and distances must be either lists or convertibl

        for axis in axes:
            self._restrictInputValue("axis", axis, AXIS_NUMBER)
        for direction in directions:
            self._restrictInputValue("direction", direction, AXIS_DIRECTION)

        global motion_completed

        motion_completed = "false"

        # Temporarily set to relative motion mode
        self.myGCode.__emit__("G91")
        while self.isReady() != "true": pass


        # Transmit move command
        command = "G0 "
        for axis, direction, distance in zip(axes, directions, distances):
            if direction == AXIS_DIRECTION.positive: distance = "" + str(distance)
            elif direction  == AXIS_DIRECTION.negative: distance = "-" + str(distance)
            command += self.myGCode.__getTrueAxis__(axis) + str(distance) + " "
        self.myGCode.__emit__(command)
        while self.isReady() != "true": pass
        return
```

Example Code

Copy Code

```
import sys
sys.path.append("..")
from _MachineMotion import *

#declare parameters for combine move
speed = 500
acceleration = 500
axesToMove = [1,2]
distances = [50, 100, 50]
directions = ["positive","positive","positive"]
mechGain = MECH_GAIN.timing_belt_150mm_turn

# Define a callback to process controller gCode responses (if desired)
def templateCallback(data):
   print ( "Controller gCode responses " + data )

mm = MachineMotion(DEFAULT_IP_ADDRESS.usb_windows, gCodeCallback = templateCallback)

mm.emitSpeed(speed)
mm.emitAcceleration(acceleration)
for axis in axesToMove:
    mm.configAxis(axis, MICRO_STEPS.ustep_8, mechGain)
mm.emitHomeAll()
mm.waitForMotionCompletion()
print("All Axes homed.")

# Simultaneously moves three axis:
#   Move axis 1 in the positive direction by 50 mm
#   Move axis 2 in the negative direction by 100 mm
#   Move axis 3 in the positive direction by 50 mm
mm.emitCombinedAxisRelativeMove(axesToMove, directions, distances)


mm.waitForMotionCompletion()
for index, axis in enumerate(axesToMove):
    print("Axis " + str(axis) + " moved " + str(distances[index]) + " in the " + directions[
```

NOTE:
The current speed and acceleration settings are applied to the combined motion of the axes.

emitDwell (*miliseconds*)

Description

Pauses motion for a specified time. This function is non-blocking; your program may accomplish other tasks while the machine is dwelling.

Parameters

miliseconds (Required) `Integer` - The duration to wait in milliseconds.

Return Value

None

Source Code

```
    def emitDwell(self, milliseconds):
        self.myGCode.__emit__("G4 P"+str(milliseconds))
        while self.isReady() != "true": pass
```

Example Code

Copy Code

```
import sys
sys.path.append("..")
from _MachineMotion import *

#Declare parameters for g-Code command
speed = 1000
acceleration = 2000
mechGain = MECH_GAIN.timing_belt_150mm_turn

# Define a callback to process controller gCode responses (if desired)
def templateCallback(data):
   print ( "Controller gCode responses " + data )

mm = MachineMotion(DEFAULT_IP_ADDRESS.usb_windows, gCodeCallback = templateCallback)

axis = AXIS_NUMBER.DRIVE1

mm.configAxis(axis, MICRO_STEPS.ustep_8, MECH_GAIN.timing_belt_150mm_turn)

mm.emitSpeed(speed)
mm.emitAcceleration(acceleration)
mm.emitHome(axis)
mm.configAxis(AXIS_NUMBER.DRIVE1, MICRO_STEPS.ustep_8, mechGain)
# Moves axis back and forth, waiting a specified amount of time between each move
mm.emitAbsoluteMove(axis, 100)
mm.emitDwell(250)
mm.emitAbsoluteMove(axis, 50)
mm.emitDwell(500)
mm.emitAbsoluteMove(axis, 100)
mm.emitDwell(1000)
mm.emitAbsoluteMove(axis, 50)
mm.emitDwell(5000)
print("emitDwell does not block this line from printing")
mm.waitForMotionCompletion()
print("but waitForMotionCompletion() will force python to wait the full 5000ms until emitDwel
```

NOTE:

The timer starts after all previous MachineMotion movement commands have finished execution.

emitHome (*axis*)

Description

Initiates the homing sequence for the specified axis.

Parameters

axis (Required) `Number` - The axis to be homed.

Return Value

None

Source Code

```
    def emitHome(self, axis):
        self._restrictInputValue("axis", axis, AXIS_NUMBER)

        global motion_completed

        motion_completed = "false"

        self.myGCode.__emit__("G28 " + self.myGCode.__getTrueAxis__(axis))
```

Example Code

Copy Code

```
import sys
sys.path.append("..")
from _MachineMotion import *

# Define a callback to process controller gCode responses (if desired)
def templateCallback(data):
   print ( "Controller gCode responses " + data )

mm = MachineMotion(DEFAULT_IP_ADDRESS.usb_windows, gCodeCallback = templateCallback)

axis = AXIS_NUMBER.DRIVE1

mm.configAxis(axis, MICRO_STEPS.ustep_8, MECH_GAIN.timing_belt_150mm_turn)
mm.emitHome(axis)
print ("Application Message: Axis "+ str(axis) +" is going home")
mm.waitForMotionCompletion()
print("Application Message: Axis "+ str(axis) +" is at home")
```

NOTE:
If configAxisDirection is set to "normal" on axis 1, axis 1 will home itself towards sensor 1A. If configAxisDirection is set to "reverse" on axis 1, axis 1 will home itself towards sensor 1B.

emitHomeAll ( )

Description

Initiates the homing sequence of all axes. All axes will home sequentially (Axis 1, then Axis 2, then Axis 3).

Return Value

None

Source Code

```
    def emitHomeAll(self):

        global motion_completed

        motion_completed = "false"

        self.myGCode.__emit__("G28")
```

Example Code

Copy Code

```
import sys
sys.path.append("..")
from _MachineMotion import *

# Define a callback to process controller gCode responses (if desired)
def templateCallback(data):
   print ( "Controller gCode responses " + data )

mm = MachineMotion(DEFAULT_IP_ADDRESS.usb_windows, gCodeCallback = templateCallback)

# Home All Axes Sequentially
mm.emitHomeAll()
print ("All Axes Moving Home")
mm.waitForMotionCompletion()
print("All Axes Homed")
```

emitRelativeMove (*axis, direction, distance*)

Description

Moves the specified axis the specified distance in the specified direction.

Parameters

`axis` (Required) `Integer` - The axis to move.
`direction` (Required) `String` - The direction of travel. Ex - "positive" or "negative"
`distance` (Required) `Number` - The travel distance in mm.

Return Value

None

Source Code

```
    def emitRelativeMove(self, axis, direction, distance):

        self._restrictInputValue("axis",axis, AXIS_NUMBER)
        self._restrictInputValue("direction", direction, AXIS_DIRECTION)

        global motion_completed

        motion_completed = "false"


        self.myGCode.__emit__("G91")
        while self.isReady() != "true": pass

        if direction == "positive":distance = "" + str(distance)
        elif direction  == "negative": distance = "-" + str(distance)

        # Transmit move command
        self.myGCode.__emit__("G0 " + self.myGCode.__getTrueAxis__(axis) + str(distance))
        while self.isReady() != "true": pass

        return
```

Example Code

Copy Code

```
import sys
sys.path.append("..")
from _MachineMotion import *

# Define a callback to process controller gCode responses (if desired)
def templateCallback(data):
   print ( "Controller gCode responses " + data )

mm = MachineMotion(DEFAULT_IP_ADDRESS.usb_windows, gCodeCallback = templateCallback)

#Define Relative Move Parameters
axis = 1
speed = 400
acceleration = 500
distance = 100
direction = "positive"
mechGain = MECH_GAIN.rack_pinion_mm_turn

#Load Relative Move Parameters
mm.emitSpeed(speed)
mm.emitAcceleration(acceleration)
mm.configAxis(axis, MICRO_STEPS.ustep_8, mechGain)

#Home Axis Before Move
print("Axis " + str(axis) + " moving home.")
mm.emitHome(axis)
mm.waitForMotionCompletion()
print("Axis " + str(axis) + " homed")

#Begin Relative Move
mm.emitRelativeMove(axis, direction, distance)
mm.waitForMotionCompletion()
print("Axis " + str(axis) + " moved " + str(distance) + "mm in the " + direction + " directio
```

emitSpeed (*speed, units*)

Description

Sets the global speed for all movement commands on all axes.

Parameters

speed (Required) `Number` - The global max speed in mm/min.
units (Optional. Default = UNITS_SPEED.mm_per_sec) `String` - Units for speed. Can be switched to UNITS_SPEED.mm_per_min

Return Value

None

Source Code

```
    def emitSpeed(self, speed, units = UNITS_SPEED.mm_per_sec):

        self._restrictInputValue("units", units, UNITS_SPEED)

        if units == UNITS_SPEED.mm_per_min:
            speed_mm_per_min = speed
        elif units == UNITS_SPEED.mm_per_sec:
            speed_mm_per_min = 60*speed


        self.myGCode.__emit__("G0 F" + str(speed_mm_per_min))
        while self.isReady() != "true": pass
```

Example Code

Copy Code

```
import sys
sys.path.append("..")
from  MachineMotion import *

# Define a callback to process controller gCode responses (if desired)
def templateCallback(data):
   print ( "Controller gCode responses " + data )

mm = MachineMotion(DEFAULT_IP_ADDRESS.usb_windows, gCodeCallback = templateCallback)

speed = 500      # The speed [mm/s] that all subsequent moves will move at
mm.emitSpeed(speed)
print("Global acceleration set to " + str(speed))
```

emitStop ( )

Description

Immediately stops all motion of all axes.

Return Value

None

Source Code

```
    def emitStop(self):
        global motion_completed

        motion_completed = "false"

        self.myGCode.__emit__("M410")

        # Wait and send a dummy packet to insure that other commands after the emit stop are
        time.sleep(0.500)
        self.myGCode.__emit__("G91")
        while self.isReady() != "true": pass
        self.myGCode.__emit__("G0 X0")
        while self.isReady() != "true": pass
```

Example Code

Copy Code

```
import sys
sys.path.append("..")
from _MachineMotion import *

import time

# Define a callback to process controller gCode responses (if desired)
def templateCallback(data):
   print ( "Controller gCode responses " + data )

mm = MachineMotion(DEFAULT_IP_ADDRESS.usb_windows, gCodeCallback = templateCallback)

#Define Relative Move Parameters
axis = 1
speed = 400
acceleration = 500
distance = 1000
direction = "positive"
mechGain = MECH_GAIN.rack_pinion_mm_turn

#Load Relative Move Parameters
mm.emitSpeed(speed)
mm.emitAcceleration(acceleration)
mm.configAxis(axis, MICRO_STEPS.ustep_8, mechGain)

#Home Axis Before Move
mm.emitHome(axis)
print("Axis " + str(axis) + " is going home")
mm.waitForMotionCompletion()
print("Axis " + str(axis) + " homed")

#Begin Relative Move
mm.emitRelativeMove(axis, direction, distance)
print("Axis " + str(axis) + " is moving " + str(distance) + "mm in the " + direction + " dire

#This move should take 5 (distance/speed) seconds to complete. Instead, we wait 2 seconds and
time.sleep(2)
mm.emitStop()
print("Axis " + str(axis) + " stopped.")
```

NOTE:
This function is a hard stop. It is not a controlled stop and consequently does not decelerate smoothly to a stop. Additionally, this function is not intended to serve as an emergency stop since this stop mechanism does not have safety ratings.

emitgCode (*gCode*)

Description

Executes raw gCode on the controller.

Parameters

gCode (Required) `string` - The g-code that will be passed directly to the controller.

Return Value

None

Source Code

```
    def emitgCode(self, gCode):
        global motion_completed

        motion_completed = "false"

        self.myGCode.__emit__(gCode)
```

Example Code

Copy Code

```
import sys
sys.path.append("..")
from _MachineMotion import *

#Declare parameters for g-Code command
speed = 500
acceleration = 500
mechGain = MECH_GAIN.timing_belt_150mm_turn

# Define a callback to process controller gCode responses (if desired)
def templateCallback(data):
   print ( "Controller gCode responses " + data )

mm = MachineMotion(DEFAULT_IP_ADDRESS.usb_windows, gCodeCallback = templateCallback)

mm.emitSpeed(speed)
mm.emitAcceleration(acceleration)
mm.emitHomeAll()
print("All Axes homed.")

# Use the G0 command to move both axis 1 and 2 by 50mm
gCodeCommand = "G0 X50 Y50"
mm.emitgCode(gCodeCommand)
mm.waitForMotionCompletion()
print("G-code command '" + gCodeCommand + "' completed by MachineMotion.")
```

NOTE:

All movement commands sent to the controller are by default in mm.

getCurrentPositions ( )

Description

Returns the current position of each axis.

Return Value

`Dictionary` A dictionary containing the current position of each axis.

Source Code

```
def getCurrentPositions(self):
    global waiting_current_position

    waiting_current_position = "true"
    self.myGCode.__emit__("M114")
    while self.isReady() != "true" and waiting_current_position == "true": pass

    return self.myGCode.currentPositions
```

NOTE:
This function returns the 'open loop' position of each axis. If your axis has an encoder, please use readEncoder.

getData (*key*)

Description

Retreives saved/persisted data from the MachineMotion controller (in key-data pairs). If the controller takes more than 3 seconds to return data, the function will return with a value of "Error - getData took too long" under the given key.

Parameters

key (Required) `String` - A Unique identifier representing the data to be retreived

Return Value

`Dictionary` A dictionary containing the saved data.

Source Code

```
    def getData(self, key):

        getDataAvailable = threading.Event()
        dataTimedOut = threading.Event()

        def asyncGetData(key, callback):
            #Send the request to MachineMotion
            self.mySocket.emit('getData', key)
            # On reception of the data invoke the callback function.
            self.mySocket.on('getDataResponse', callback)

        def asyncCallback(data):
            self.asyncResult = data
            getDataAvailable.set()

        def threadTimeout():
           time.sleep(3)
           dataTimedOut.set()

        #timer here to force call asyncCallback on timeout #kill timer#
        # If this fails, return a key value pair - key is 'error' value is description why er
        getDataThread = threading.Thread(target = asyncGetData, args=(key, asyncCallback,))
        timeoutThread = threading.Thread(target = threadTimeout)

        getDataThread.start()
        timeoutThread.start()
        while True:
            if getDataAvailable.isSet():
                getDataResult = json.loads("".join(self.asyncResult))
                return getDataResult
            elif dataTimedOut.isSet():
                getDataResult = json.loads('{"data":"Error - getData took too long"}')
                return getDataResult
```

Example Code

Copy Code

```
import sys
sys.path.append("..")
from _MachineMotion import *

mm = MachineMotion(DEFAULT_IP_ADDRESS.usb_windows)

# Initialize the key-value pair to save on the controller
dataKey = "example_key"
dataContent = "save_this_string_on_the_controller"

# Saving a string on the controller
mm.saveData(dataKey, dataContent)
print("Data: '" + dataContent + "' saved on controller under key '" + dataKey + "'")

# Retreiving data from the controller
retrievedData = mm.getData(dataKey)
print("Data: '" + retrievedData['data'] + "' retrieved from controller using key '" + dataKey
```

readEncoder(*encoder, readingType*)

Description

Returns the last received encoder position in counts.

Parameters

encoder (Required) `Integer` - The identifier of the encoder to read
readingType (Required) `String` - Either 'real time' or 'stable'. In 'real time' mode, readEncoder will return the most recently received encoder information. In 'stable' mode, readEncoder will update its return value only after the encoder output has stabilized around a specific value, such as when the axis has stopped motion.

Return Value

`Integer` The current position of the encoder, in counts. The encoder has 3600 counts per revolution.

Source Code

```
    def readEncoder(self, encoder, readingType="realTime"):
        self._restrictInputValue("readingType", readingType, ENCODER_TYPE)

        if (self.isEncoderIdValid( encoder ) == False):
            print ( "DEBUG: unexpected encoder identifier: encoderId= " + str(encoder) )
            return

        if readingType == ENCODER_TYPE.real_time:
            self.myEncoderStablePositions
            return self.myEncoderRealtimePositions[encoder]
        elif readingType == ENCODER_TYPE.stable:
            return self.myEncoderStablePositions[encoder]

        return
```

Example Code

Copy Code

```
import sys
sys.path.append("..")
from _MachineMotion import *

mm = MachineMotion(DEFAULT_IP_ADDRESS.usb_windows)

#Adjust this line to match whichever AUX port the encoder is plugged into
encoderPort = AUX_PORTS.aux_1
encoderPortLabel = {AUX_PORTS.aux_1: "AUX 1", AUX_PORTS.aux_2: "AUX 2", AUX_PORTS.aux_3: "AUX

checkInput = True
print("Reading and printing encoder output for 10 seconds:")
for i in range(1,30):
   realTimeOutput = mm.readEncoder(encoderPort, ENCODER_TYPE.real_time)
   stableOutput = mm.readEncoder(encoderPort, ENCODER_TYPE.stable)
   print("Encoder on port " + encoderPortLabel[encoderPort] + "\t Realtime Position =" + str(
   time.sleep(.25)

   if realTimeOutput != 0:
      checkInput = False

if(checkInput):
   print("The encoder is not receiving any data. Please check the following: \n\t Is the enco
```

NOTE:

The encoder position returned by this function may be delayed by up to 250 ms due to internal propogation delays.

saveData (*key, data*)

Description

Saves/persists data within the MachineMotion Controller in key - data pairs.

Parameters

key (Required) `String` - A string that uniquely identifies the data to save for future retreival.
data (Required) `String` - The data to save to the machine. The data must be convertible to JSON format.

Return Value

None

Source Code

```
    def saveData(self, key, data):

        # Create a new object and augment it with the key value.
        dataPack = {}
        dataPack["fileName"] = key
        dataPack["data"] = data

        # Send the request to MachineMotion
        self.mySocket.emit('saveData', json.dumps(dataPack))
        time.sleep(0.05)
```

Example Code

Copy Code

```
import sys
sys.path.append("..")
from _MachineMotion import *

mm = MachineMotion(DEFAULT_IP_ADDRESS.usb_windows)

# Initialize the key-value pair to save on the controller
dataKey = "example_key"
dataContent = "save_this_string_on_the_controller"

# Saving a string on the controller
mm.saveData(dataKey, dataContent)
print("Data: '" + dataContent + "' saved on controller under key '" + dataKey + "'")

# Retreiving data from the controller
retrievedData = mm.getData(dataKey)
print("Data: '" + retrievedData['data'] + "' retrieved from controller using key '" + dataKey
```

NOTE:
The Data continues to exist even when the controller is shut off. However, writing to a previously used key will override the previous value.

setPosition (*axis, position*)

Description

Override the current position of the specified axis to a new value.

Parameters

`axis` (Required) `Number` - Overrides the position on this axis.
`position` (Required) `Number` - The new position value in mm.

Return Value

None

Source Code

```
    def setPosition(self, axis, position):
        self._restrictInputValue("axis", axis, AXIS_NUMBER)

        # Transmit move command
        self.myGCode.__emit__("G92 " + self.myGCode.__getTrueAxis__(axis) + str(position))
        while self.isReady() != "true": pass
```

Example Code

Copy Code

```
import sys
sys.path.append("..")
from _MachineMotion import *

mm = MachineMotion(DEFAULT_IP_ADDRESS.usb_windows)

axis = 1                                    #The axis that you'd like to move
speed = 400                                 #The max speed you'd like to move at
acceleration = 500                          #The constant acceleration and decceleration valu
position = 100                              #The absolute position you'd like to move to
mechGain = MECH_GAIN.rack_pinion_mm_turn    #The mechanical gain of the actuator on the axis

mm.emitSpeed(speed)
mm.emitAcceleration(acceleration)
mm.configAxis(axis, MICRO_STEPS.ustep_16, mechGain)

mm.emitHome(axis)
mm.waitForMotionCompletion()
print("Axis " + str(axis) + " homed")

print("Absolute Moves are referenced from home")
mm.emitAbsoluteMove(axis, position)
mm.waitForMotionCompletion()
print("Axis " + str(axis) + " is " + str(position) + "mm away from home")

mm.setPosition(axis, 0)
print("Absolute moves on axis " + str(axis) + " are now referenced from " +  str(position) +
time.sleep(2)
position2 = 30
print("Now moving to absolute Position " + str(position2) + " mm, referenced from location '
mm.emitAbsoluteMove(axis, position2)
mm.waitForMotionCompletion()
print("Axis " + str(axis) + " is now " + str(position2) + "mm from reference position and " +
```

waitForMotionCompletion ( )

Description

Pauses python program execution until machine has finished its current movement.

Return Value

None

Source Code

```
    def waitForMotionCompletion(self):
        global waiting_motion_status

        waiting_motion_status = "true"
        self.emitgCode("V0")
        while  self.isMotionCompleted() != "true": pass
```

Example Code

Copy Code

```
import sys
sys.path.append("..")
from _MachineMotion import *

# Define a callback to process controller gCode responses (if desired)
def templateCallback(data):
   print ( "Controller gCode responses " + data )

mm = MachineMotion(DEFAULT_IP_ADDRESS.usb_windows, gCodeCallback = templateCallback)

mm.emitHome(1)

# Move the axis one to position 100 mm
mm.emitAbsoluteMove(1, 100)
print("This message gets printed immediately")
mm.waitForMotionCompletion()
print("This message gets printed once machine is finished moving")
```