Python

Python Homepage - download Python here.

Many users want to interact with ASI devices through Python. There are at least 3 viable options:

1. Send serial commands directly using the pyserial library
2. Use pymmcore package to access the MMCore layer of Micro-Manager where ASI devices have excellent support
3. Use Pycro-manager to access almost all of Micro-Manager's capability including the MMCore layer

Via pyserial library

We use the pyserial library and Python 3 in-house. You can install pyserial with “pip install pyserial” in the terminal. See separate documentation of serial commands, e.g. via the quick start page or detailed documentation of serial commands.

You can adapt this script to work with Tiger devices, you will need to send the card address before some serial commands. For example, with a MS2000 you would send `m x=1000 y=1000`, and on Tiger you would send `2m x=1000 y=1000` provided the card address for the device is 2.

This script works best when the baud rate is set to 115200.

Here is an example script:

Last tested on Windows 10 64-Bit, Python 3.10.1 64-Bit, and pyserial 3.5.

This class manages the serial connection.

serialport.py

```
from serial import Serial
from serial import SerialException
from serial import EIGHTBITS
from serial import PARITY_NONE
from serial import STOPBITS_ONE
from serial.tools import list_ports

class SerialPort:
    """
    A utility class for managing a RS232 serial connection using
the pyserial library.

    """

    def __init__(self, com_port: str, baud_rate: int, report: bool
= True):
        self.serial_port = Serial()
```

```
        self.com_port = com_port
        self.baud_rate = baud_rate
        # user feedback settings
        self.report = report
        self.print = self.report_to_console

    @staticmethod
    def scan_ports() -> list[str]:
        """Returns a sorted list of COM ports."""
        com_ports = [port.device for port in
list_ports.comports()]
        com_ports.sort(key=lambda value: int(value[3:]))
        return com_ports

    def connect_to_serial(self, rx_size: int = 12800, tx_size: int
= 12800, read_timeout: int = 1, write_timeout: int = 1) -> None:
        """Connect to the serial port."""
        # serial port settings
        self.serial_port.port = self.com_port
        self.serial_port.baudrate = self.baud_rate
        self.serial_port.parity = PARITY_NONE
        self.serial_port.bytesize = EIGHTBITS
        self.serial_port.stopbits = STOPBITS_ONE
        self.serial_port.xonoff = False
        self.serial_port.rtscts = False
        self.serial_port.dsrdtr = False
        self.serial_port.write_timeout = write_timeout
        self.serial_port.timeout = read_timeout

        # set the size of the rx and tx buffers before calling
open
        self.serial_port.set_buffer_size(rx_size, tx_size)

        # try to open the serial port
        try:
            self.serial_port.open()
        except SerialException:
            self.print(f"SerialException: can't connect to
{self.com_port} at {self.baud_rate}!")

        if self.is_open():
            # clear the rx and tx buffers
            self.serial_port.reset_input_buffer()
            self.serial_port.reset_output_buffer()
            # report connection status to user
            self.print("Connected to the serial port.")
            self.print(f"Serial port = {self.com_port} :: Baud
rate = {self.baud_rate}")

    def disconnect_from_serial(self) -> None:
        """Disconnect from the serial port if it's open."""
```

```
        if self.is_open():
            self.serial_port.close()
            self.print("Disconnected from the serial port.")

    def is_open(self) -> bool:
        """Returns True if the serial port exists and is open."""
        # short circuits if serial port is None
        return self.serial_port and self.serial_port.is_open

    def report_to_console(self, message: str) -> None:
        """Print message to the output device, usually the
console."""
        # useful if we want to output data to something other than
the console (ui element etc)
        if self.report:
            print(message)

    def send_command(self, cmd: bytes) -> None:
        """Send a serial command to the device."""
        # always reset the buffers before a new command is sent
        self.serial_port.reset_input_buffer()
        self.serial_port.reset_output_buffer()
        # send the serial command to the controller
        command = bytes(f"{cmd}\r", encoding="ascii")
        self.serial_port.write(command)
        self.print(f"Send: {command.decode(encoding='ascii')}")

    def read_response(self) -> str:
        """Read a line from the serial response."""
        response = self.serial_port.readline()
        response = response.decode(encoding="ascii")
        self.print(f"Recv: {response.strip()}")
        return response # in case we want to read the response
```

The MS2000 class is a subclass of SerialPort and adds input validation to the constructor.

ms2k.py

```
from serialport import SerialPort

class MS2000(SerialPort):
    """
    A utility class for operating the MS2000 from Applied
Scientific Instrumentation.

    Move commands use ASI units: 1 unit = 1/10 of a micron.
    Example: to move a stage 1 mm on the x axis, use
self.moverel(10000)
```

```
    Manual:
        http://asiimaging.com/docs/products/ms2000

    """

    # all valid baud rates for the MS2000
    # these rates are controlled by dip switches
    BAUD_RATES = [9600, 19200, 28800, 115200]

    def __init__(self, com_port: str, baud_rate: int=115200,
report: str=True):
        super().__init__(com_port, baud_rate, report)
        # validate baud_rate input
        if baud_rate in self.BAUD_RATES:
            self.baud_rate = baud_rate
        else:
            raise ValueError("The baud rate is not valid. Valid
rates: 9600, 19200, 28800, or 115200.")

    # ------------------------------ #
    #     MS2000 Serial Commands     #
    # ------------------------------ #

    def moverel(self, x: int=0, y: int=0, z: int=0) -> None:
        """Move the stage with a relative move."""
        self.send_command(f"MOVREL X={x} Y={y} Z={z}\r")
        self.read_response()

    def moverel_axis(self, axis: str, distance: int) -> None:
        """Move the stage with a relative move."""
        self.send_command(f"MOVREL {axis}={distance}\r")
        self.read_response()

    def move(self, x: int=0, y: int=0, z: int=0) -> None:
        """Move the stage with an absolute move."""
        self.send_command(f"MOVE X={x} Y={y} Z={z}\r")
        self.read_response()

    def move_axis(self, axis: str, distance: int) -> None:
        """Move the stage with an absolute move."""
        self.send_command(f"MOVE {axis}={distance}\r")
        self.read_response()

    def set_max_speed(self, axis: str, speed:int) -> None:
        """Set the speed on a specific axis. Speed is in mm/s."""
        self.send_command(f"SPEED {axis}={speed}\r")
        self.read_response()

    def get_position(self, axis: str) -> int:
        """Return the position of the stage in ASI units (tenths
```

```
of microns)."""
        self.send_command(f"WHERE {axis}\r")
        response = self.read_response()
        return int(response.split(" ")[1])

    def get_position_um(self, axis: str) -> float:
        """Return the position of the stage in microns."""
        self.send_command(f"WHERE {axis}\r")
        response = self.read_response()
        return float(response.split(" ")[1])/10.0

    # ------------------------------ #
    #    MS2000 Utility Functions    #
    # ------------------------------ #

    def is_axis_busy(self, axis: str) -> bool:
        """Returns True if the axis is busy."""
        self.send_command(f"RS {axis}?\r")
        return "B" in self.read_response()

    def is_device_busy(self) -> bool:
        """Returns True if any axis is busy."""
        self.send_command("/")
        return "B" in self.read_response()

    def wait_for_device(self, report: bool = False) -> None:
        """Waits for the all motors to stop moving."""
        if not report:
            print("Waiting for device...")
        temp = self.report
        self.report = report
        busy = True
        while busy:
            busy = self.is_device_busy()
        self.report = temp
```

And using the class in a script:

main.py

```
from ms2k import MS2000

def main():
    # scan system for com ports
    print(f"COM Ports: {MS2000.scan_ports()}")

    # connect to the MS2000
    ms2k = MS2000("COM9", 115200)
```

```
    ms2k.connect_to_serial()
    if not ms2k.is_open():
        print("Exiting the program...")
        return

    # move the stage
    ms2k.moverel(10000, 0)
    ms2k.wait_for_device()
    ms2k.moverel(0, 10000)
    ms2k.wait_for_device()
    ms2k.moverel(-10000, 0)
    ms2k.wait_for_device()
    ms2k.moverel(0, -10000)
    ms2k.wait_for_device()

    # close the serial port
    ms2k.disconnect_from_serial()

if __name__ == "__main__":
    main()
```

Note: Make sure you enter the correct COM port and baud rate in the constructor for the MS2000 class.

Via pymmcore

See the pymmcore GitHub page for information.

Micro-Manager has excellent support for ASI devices built via “device adapters” which are part of the MMCore layer of Micro-Manager which are exposed in Python using the pymmcore library. You can use Micro-Manager's hardware control APIs (e.g. to move stages) and also Micro-Manager's device properties. These properties include a mechanism for sending arbitrary serial commands.

To run the example below, you will need to make a configuration file using the `Hardware Configuration Wizard`. You can use either the `ASITiger` or `ASIStage` device adapter and add the `XYStage` to your hardware configuration. Save the configuration file as `pymmcore_test.cfg`.

An example script to control an XYStage

```
import pymmcore
import os

def main():
    hardware_cfg = "pymmcore_test.cfg"
    mm_directory = "C:/Program Files/Micro-Manager-2.0"

    version_info = pymmcore.CMMCore().getAPIVersionInfo()
```

```
    print(version_info)

    mmc = pymmcore.CMMCore()
    print(mmc.getVersionInfo())

    # load the device adapters
    mmc.setDeviceAdapterSearchPaths([mm_directory])
    mmc.loadSystemConfiguration(os.path.join(mm_directory,
hardware_cfg))

    # get the stage device name
    xy_stage = mmc.getXYStageDevice()
    print(f"XYStage Device: {xy_stage}")

    # move the xy stage
    mmc.setRelativeXYPosition(10000, 0)
    mmc.waitForDevice(xy_stage)

    mmc.setRelativeXYPosition(0, 10000)
    mmc.waitForDevice(xy_stage)

    mmc.setRelativeXYPosition(-10000, 0)
    mmc.waitForDevice(xy_stage)

    mmc.setRelativeXYPosition(0, -10000)
    mmc.waitForDevice(xy_stage)

if __name__ == "__main__":
    main()
```

Via Pycro-Manager

See Pycro-Manager documentation.

Micro-Manager which has excellent support for ASI devices built in. Using Pycro-Manager is especially valuable if you also have non-ASI devices to control that are supported in Micro-Manager. Instead of sending serial commands directly you use Micro-Manager's hardware control APIs (e.g. to move stages) and also Micro-Manager's device properties. These properties include a mechanism for sending arbitrary serial commands.

python, serial