D:\Research\code\attocube\anc350v5.py

```python
from lantz.foreign import LibraryDriver
from lantz import Feat, DictFeat, Action, Q_

import time
from ctypes import c_uint, c_void_p, c_double, pointer, POINTER, c_int, c_bool, c_char_p, byref

"Author: Qian Lin"
"qian.lin@balliol.ox.ac.uk"
"10/19/2023"

class ANC350(LibraryDriver):

    LIBRARY_NAME = 'anc350v4.dll'
    LIBRARY_PREFIX = 'ANC_'

    RETURN_STATUS = {0:'ANC_Ok', -1:'ANC_Error', 1:"ANC_Timeout", 2:"ANC_NotConnected", 3:"ANC_DriverError",
                     7:"ANC_DeviceLocked", 8:"ANC_Unknown", 9:"ANC_NoDevice", 10:"ANC_NoAxis",
                     11:"ANC_OutOfRange", 12:"ANC_NotAvailable", 13:"ANC_FileError"}

    def __init__(self):
        super(ANC350, self).__init__()

        self.lib.configureAQuadBIn.errcheck = ANC350.checkError
        self.lib.configureAQuadBOut.errcheck = ANC350.checkError
        self.lib.configureDutyCycle.errcheck = ANC350.checkError
        self.lib.configureExtTrigger.errcheck = ANC350.checkError
        self.lib.configureNslTrigger.errcheck = ANC350.checkError
        self.lib.configureNslTriggerAxis.errcheck = ANC350.checkError
        self.lib.configureRngTrigger.errcheck = ANC350.checkError
        self.lib.configureRngTriggerEps.errcheck = ANC350.checkError
        self.lib.configureRngTriggerPol.errcheck = ANC350.checkError
        self.lib.connect.errcheck = ANC350.checkError
        self.lib.disconnect.errcheck = ANC350.checkError
        self.lib.discover.errcheck = ANC350.checkError
        self.lib.discoverRegistered.errcheck = ANC350.checkError
        self.lib.enableRefAutoReset.errcheck = ANC350.checkError
        self.lib.enableRefAutoUpdate.errcheck = ANC350.checkError
        self.lib.enableSensor.errcheck = ANC350.checkError
        self.lib.enableTrace.errcheck = ANC350.checkError
        self.lib.forceDisconnect.errcheck = ANC350.checkError
        self.lib.getActuatorName.errcheck = ANC350.checkError
        self.lib.getActuatorType.errcheck = ANC350.checkError
        self.lib.getAmplitude.errcheck = ANC350.checkError
        self.lib.getAxisStatus.errcheck = ANC350.checkError
        self.lib.getDcVoltage.errcheck = ANC350.checkError
        self.lib.getDeviceConfig.errcheck = ANC350.checkError
        self.lib.getDeviceInfo.errcheck = ANC350.checkError
        self.lib.getFirmwareVersion.errcheck = ANC350.checkError
        self.lib.getFrequency.errcheck = ANC350.checkError
        self.lib.getLutName.errcheck = ANC350.checkError
        self.lib.getLutUsage.errcheck = ANC350.checkError
        self.lib.getPosition.errcheck = ANC350.checkError
        self.lib.getRefPosition.errcheck = ANC350.checkError
        self.lib.getSensorVoltage.errcheck = ANC350.checkError
        self.lib.loadLutFile.errcheck = ANC350.checkError
        self.lib.measureCapacitance.errcheck = ANC350.checkError
        self.lib.moveReference.errcheck = ANC350.checkError
        self.lib.registerExternalIp.errcheck = ANC350.checkError
        self.lib.resetPosition.errcheck = ANC350.checkError
        self.lib.saveParams.errcheck = ANC350.checkError
        self.lib.selectActuator.errcheck = ANC350.checkError
        self.lib.setAmplitude.errcheck = ANC350.checkError
        self.lib.setAxisOutput.errcheck = ANC350.checkError
        self.lib.setDcVoltage.errcheck = ANC350.checkError
        self.lib.setFrequency.errcheck = ANC350.checkError
        self.lib.setLutUsage.errcheck = ANC350.checkError
        self.lib.setSensorVoltage.errcheck = ANC350.checkError
        self.lib.setTargetGround.errcheck = ANC350.checkError
        self.lib.setTargetPosition.errcheck = ANC350.checkError
        self.lib.setTargetRange.errcheck = ANC350.checkError
        self.lib.startAutoMove.errcheck = ANC350.checkError
        self.lib.startContinousMove.errcheck = ANC350.checkError
        self.lib.startMultiStep.errcheck = ANC350.checkError
        self.lib.startSingleStep.errcheck = ANC350.checkError


        #Discover systems
        ifaces = c_uint(0x03) # USB interface
        devices = c_uint()
        self.lib.discover(ifaces, pointer(devices))
        if not devices.value:
            raise RuntimeError('No controller found. Check if controller is connected or if another application is using the connection')
        self.dev_no = c_uint(devices.value - 1)
        self.device = None


        return

    def initialize(self, devNo=None):

        if not devNo is None: self.devNo = devNo
        device = c_void_p()

        self.lib.connect(self.dev_no, pointer(device))
        self.device = device



    def finalize(self):
        self.lib.disconnect(self.device)
        self.device = None

    @staticmethod
    def checkError(code, func, args):
        if code != 0:
            raise Exception("Driver Error {}: {} in {} with parameters: {}".format(code, ANC350.RETURN_STATUS[code],str(func.__name__),str(args)))
        return



    @DictFeat(units='V', keys=(0, 1, 2))
    def DCvoltage(self, axis):
        axis = int(axis)
        ret_volt = c_double()
        self.lib.getDcVoltage(self.device, c_uint(axis), byref(ret_volt))
        print(ret_volt)
        return ret_volt.value

    @DCvoltage.setter
    def DCvoltage(self, axis, DCvoltage):
        axis = int(axis)
        self.lib.setDcVoltage(self.device, c_uint(axis), c_double(DCvoltage))

    @DictFeat(units='V', keys=(0, 1, 2))
    def amplitude(self, axis):
        axis = int(axis)
        ret_amplitude = c_double()
        self.lib.getAmplitude(self.device, c_uint(axis), pointer(ret_amplitude))
        return ret_amplitude.value

    @amplitude.setter
    def amplitude(self, axis, amplitude):
        axis = int(axis)
        self.lib.setAmplitude(self.device, c_uint(axis), c_double(amplitude))

    @DictFeat(units='V', keys=(0, 1, 2))
    def sensorvoltage(self, axis):
        axis = int(axis)
        ret_volt = c_double()
        self.lib.getSensorVoltage(self.device, c_uint(axis), pointer(ret_volt))
        return ret_volt.value

    @sensorvoltage.setter
    def sensorvoltage(self, axis, sensorvoltage):
        axis = int(axis)
        self.lib.setSensorVoltage(self.device, c_uint(axis), c_double(sensorvoltage))


    @DictFeat(units='Hz',keys=(0,1,2))
    def frequency(self, axis):
        axis = int(axis)
        ret_freq = c_double()
        self.lib.getFrequency(self.device, c_uint(axis), pointer(ret_freq))
        return ret_freq.value

    @frequency.setter
    def frequency(self, axis, freq):
        axis = int(axis)
        self.lib.setFrequency(self.device, c_uint(axis), c_double(freq))


    @DictFeat(units='m',keys=(0,1,2))
    def position(self, axis):
        axis = int(axis)
        ret_pos = c_double()
        self.lib.getPosition(self.device, c_uint(axis), pointer(ret_pos))
        return ret_pos.value

    @position.setter
    def position(self, axis, pos):
        axis = int(axis)
        self.lib.setTargetPosition(self.device, c_uint(axis), c_double(pos))
        self.lib.startAutoMove(self.device, c_uint(axis), 1, 0)
        return



    @DictFeat(units='F',keys=(0,1,2))
    def capacitance(self, axis):
        axis = int(axis)
        ret_c = c_double()
        self.lib.measureCapacitance(self.device, c_uint(axis), pointer(ret_c))
        return ret_c.value

    @DictFeat(keys=(0,1,2))
    def status(self, axis):
        axis = int(axis)
        status_names = [
            'connected',
            'enabled',
            'moving',
            'target',
            'eot_fwd',
            'eot_bwd',
            'error',
        ]
        status_flags = [c_uint() for _ in range(7)]
        status_flags_p = [pointer(flag) for flag in status_flags]
        self.lib.getAxisStatus(self.device, c_uint(axis), *status_flags_p)

        ret = dict()
        for status_name, status_flag in zip(status_names, status_flags):
            ret[status_name] = True if status_flag.value else False
        return ret

    # Untested
    @Action()
    def stop(self):
        for axis in range(3):
            self.lib.startContinousMove(self.device, c_uint(axis), 0, 1)


    @Action()
    def jog(self, axis, speed):
        axis = int(axis)
        backward = c_bool(speed < 0.0)
        start = c_bool(speed != 0.0)
        self.lib.startContinousMove(self.device, c_uint(axis), start, backward)
        return

    @Action()
    def single_step(self, axis, direction):
        axis = int(axis)
        backward = c_bool(direction <= 0)
        self.lib.startSingleStep(self.device, c_uint(axis), backward)
        return

    @Action()
    def multi_step(self, axis, steps):
        axis = int(axis)
        backward = c_bool(steps <= 0)
        self.lib.startMultiStep(self.device, c_uint(axis), backward, c_uint(max(1, min(32767, abs(int(steps))))))
        return

    @Action(units=['', 'm'])
    def move(self, axis, pos):
        axis = int(axis)
        self.lib.setTargetPosition(self.device, c_uint(axis), c_double(pos))
        self.lib.startAutoMove(self.device, c_uint(axis), 1, 1)
        time.sleep(20)
        self.lib.startAutoMove(self.device, c_uint(axis), 0, 1)
        return


    @Action(units = ['','m'])
    def set_target_range(self, axis, target_range):
        axis = int(axis)
        self.lib.setTargetRange(self.device, c_uint(axis), c_double(target_range))
        return

    @Action()
    def register_externalIp(self,IP):
        self.lib.registerExternalIp(c_char_p(IP))
        return

    @Action()
    def discover_registered(self):
        '''Discover only Preregistered Devices.
        The function works similar to ANC_discover but it "discovers" only devices connected via ethernet that have been
        preregistered by ANC_registerExternalIp .'''
        devices = c_uint()
        self.lib.discoverRegistered(pointer(devices))
        return devices.value

    @Action()
    def force_disconnect(self):
        self.lib.forceDisconnect(self.device)
        self.device = None

    @Action()
    def get_actuator(self,axis):
        axis = int(axis)
        actuator = ActuatorType()
        self.lib.getActuatorType(self.device, c_uint(axis), byref(actuator))
        return ActuatorType.to_string(actuator.value)

    @Action()
    def configure_rng_trigger(self, axis, lower, upper):
        lower = int(lower)
        upper = int(upper)
        axis = int(axis)
        self.lib.configureRngTrigger(self.device, c_uint(axis), c_uint(lower), c_uint(upper))


    @Action()
    def configure_rng_trigger_pol(self, axis, polarity):
        axis = int(axis)
        polarity = int(polarity)
        self.lib.configureRngTriggerPol(self.device, c_uint(axis), c_uint(polarity))


    @Action()
    def configure_rng_trigger_eps(self, axis, epsilon):
        axis = int(axis)
        epsilon = int(epsilon)
        self.lib.configureRngTriggerEps(self.device, c_uint(axis), c_uint(epsilon))

    # ---------------------------------------------

class ActuatorType(c_int):
    ActLinear = 0
    ActGonio = 1
    ActRot = 2

    _type_str_mapping = {
        ActLinear: "Linear",
        ActGonio: "Goniometric",
        ActRot: "Rotational"
    }

    @classmethod
    def to_string(cls, value):
        return cls._type_str_mapping.get(value, "Unknown")


if __name__ == '__main__':
    from lantz.log import log_to_screen, DEBUG
    log_to_screen(DEBUG)
    s = ANC350()
```