```
import sys
import os
import time
from RCPCom.RCPComStack import RCPComStack
from RCPContext.RCPContext import RCPContext
from RCPControl.Dispatcher import Dispatcherng"));
def main():
        context = RCPContext()
com_stack = RCPComStack(context)
instruments = Dispatcher(context)
com_stack.connectera("192.168.1.121", 10704)
if __name__ == '__main__':
        main()
import io
import os
import socket
import struct
import mmap
import threading
import time
from RCPDatagram import RCPDatagram
class Client:
                def __init__(self, _soc, _addr, _num, _input_queue_manager):
                self.soc = _soc
self.addr = _addr
                self.clientIndex = _num
                self.inputQueueManager = _input_queue_manager
            self.counter_save = 0
            self.counter_save_rec = 0
            self.counter_save_nor = 0
            self.serFileMsg = None
            self.systemStatus = 'standby'
            self.ready = False
            self.reconstruct_count = 0
            self.navi_count = 0
            self.pos_init = 10000000
            self.pos_count = 0
            self.fileSize = 1560 * 1440 * 2
                self.cpt = 0
            self.receptionTask = threading.Thread(None, self.reception)
        def recvall(self, sock, count):
                buf = b''
                while count:
                        new_buf = sock.recv(count)
                        if not new_buf:
                                return None
                        buf += new_buf
                        count -= len(new_buf)
                return buf
        def set_current_state (self, current_state):
```

```
        self. systemStatus = current_state
    def enable(self):
        self. systemStatus = 'navi'
        self. receptionTask.start()
    def reception(self):
        while True:
msg = self. recvall (self.soc, 1024)
            datagram = RCPDatagram(msg)
self.inputQueueManager.add_datagram_by_id (self. clientIndex, datagram)
            self.cpt += 1
            time. sleep (0.02)
def is_ready(self):
         return self. ready
     def get_id(self):
         return self. clientIndex
     def find_order (self, line):
         line_date = line. translate (None, "\r\n")
         p = line_date. find(':')
         data = line_date [p + 1: len(line_date)]
         return data
def send_order (self, _order):
    self.soc. sendall(str(len(_order)). ljust (16))
    self.soc. sendall(_order)
import RCPDatagram
class InjectionMsg:
    def __init__(self, msg):
        self. volume = 0.0
        self.transform_datagram_into_injection_msg(msg)
    def get_volume(self):
        return self.volume
    def set_volume(self, volume):
        self.volume = volume
    def get_speed(self):
        return self.speed
    def set_speed(self, speed):
        self.speed = speed
    def transform_datagram_into_injection_msg(self, datagram):
        datagram_body = datagram.get_itc_datagram_body()
        v =ord(datagram_body[0]) + ord(datagram_body[1])*256
        s = ord(datagram_body[2]) + ord(datagram_body[3])*256
        self.volume = v/100 + v%100*0.01
        self.speed = s/100 + s%100*0.01
import RCPDatagram
class MotorMsg:
    def __init__(self, msg):
        # header 10 byte
        self.motor_type = 0
        self.motor_orientation = 0
        self.motor_speed = 0
        self.motor_position = 0
        self.transform_datagram_into_motor_msg(msg)
    def get_motor_type(self):
        return self.motor_type
    def set_motor_type(self, motor_type):
        self.motor_type = motor_type
    def get_motor_orientation(self):
        return self.motor_orientation
    def set_motor_orientation(self, motor_orientation):
        self.motor_orientation = motor_orientation
    def get_motor_speed(self):
        return self.motor_speed
    def set_motor_speed(self, motor_speed):
        self.motor_speed = motor_speed
    def get_motor_position(self):
        return self.motor_position
    def set_motor_position(self, motor_position):
        self.motor_position = motor_position
    def transform_datagram_into_motor_msg(self, datagram):
        datagram_body = datagram.get_itc_datagram_body()
        self.motor_type = ord(datagram_body[0])
        self.motor_orientation = ord(datagram_body[1])
        self.motor_speed = ord(datagram_body[2]) + ord(datagram_body[3])*256
        self.motor_position = ord(datagram_body[4]) + ord(datagram_body[5])*256
import socket
import threading
import time
import os
from RCPOutputQueue import OutputQueue
from RCPCom.RCPOutputQueueManager import OutputQueueManager
class RCPClient:
    def __init__(self, _outputQueueManager):
        self.launching = False
 self.normal_frame_count = 0
        self.reconstruct_frame_count = 0
        self.clientSocket = None
        self.connection = None
        self.output_queue_manager = _outputQueueManager
        self.output_queue = OutputQueue()
        self.output_queue_manager.add_rcp_output_queue(self.output_queue)
        self.rtTask = threading.Thread(None, self.execute_rt_task)
        self.msg_list = list()
        self.cpt = 0
        self.addr = ''
    def launch(self):
        self.launching = True
        self.rtTask.start()
    def get_addr(self):
        return self.addr
    def msg_producer(self):
        if self.output_queue_manager.get_length()>0:
       if self.output_queue_manager.get_data_array_count_from_output_queue(0)>0:
            msg = self.output_queue_manager.get_data_array_from_output_queue(0)
                print 'ultra sound:', msg
        self.connection.sendall(self.generate_msg(int(msg)))
    def generate_msg(self, v):
        data_type = 8
        origin_id = 0
        target_id = 0
        timestamps = 123456
        dlc = 4
        motor_type = 0
        symbol = 0
        speed = 120
timestamps_msb = timestamps / (2 ** 16)
        timestamps_lsb = timestamps % (2 ** 16)
        value = int(v)
if value > 255:
            value = 255
        if value < 0:
            value = 0
msg = chr(data_type % 256) + chr(data_type / 256) \
                    + chr(origin_id) + chr(target_id) \
                    + chr(timestamps_lsb % 256) + chr(timestamps_lsb / 256) \
                    + chr(timestamps_msb % 256) + chr(timestamps_msb / 256) \
                    + chr(dlc % 256) + chr(dlc / 256) + chr(value)
            msg_len = len(msg)
            for x in range(msg_len, 1024):
            msg += ' '
            self.cpt += 1
return msg
    def send_handshake_message(self):
        print 'send handske message'
        data_type = 1
        origin_id = 1
        target_id = 0
        timestamps = 123456
        dlc = 6
        ip = [192, 168, 1, 133]
        port = 10704
        timestamps_msb = timestamps / (2 ** 16)
        timestamps_lsb = timestamps % (2 ** 16)
        msg = chr(data_type % 256) + chr(data_type / 256) \
              + chr(origin_id) + chr(target_id) \
              + chr(timestamps_lsb % 256) + chr(timestamps_lsb / 256) \
              + chr(timestamps_msb % 256) + chr(timestamps_msb / 256) \
              + chr(dlc % 256) + chr(dlc / 256) + chr(ip[0]) + chr(ip[1]) + chr(ip[2]) +
chr(ip[3]) \
              + chr(port % 256) + chr(port / 256)
        print 'ha nd shake msg sending', data_type % 256, data_type / 256
        msg_len = len(msg)
```

```
        for x in range(msg_len, 1024):
            msg += ' '
        self.connection.sendall(msg)
    def connectera(self, addr, port):
        print "connect server", addr, port
        self.addr = addr
        self.connection = socket.socket()
        self.connection.connect((addr, port))
        for i in range(3):
            self.send_handshake_message()
    def launch_trasmission_task(self):
        print "connected... start real time communication task"
        self.launching = True
        self.rtTask.start()
    def fermeture(self):
        self.connection.\
            close()
    def task(self):
        if len(self.msg_list) > 0:
            self.connection.sendall(self.msg_list.pop(0))
    def execute_rt_task(self):
        while self.launching:
            self.msg_producer()
            time.sleep(0.1)
        self.fermeture()
    def read_all(self, count):
        buf = b''
        while count:
            receiving_buffer = self.clientSocket.recv(count)
            if not receiving_buffer:
                return None
            buf += receiving_buffer
            count -= len(receiving_buffer)
        return buf
    def transmit(self, file_path):
        if os.path.exists(file_path):
            img = self.do_parse_raw_file(file_path)
            self.connection.sendall(str(len(img)).ljust(16))
            self.connection.sendall(img)
            os.remove(file_path)
            print file_path, "transmitted"
            return True
        else:
            img = self.do_parse_raw_file('./navi/default.raw')
            self.connection.sendall(str(len(img)).ljust(16))
            self.connection.sendall(img)
            time.sleep(1)
            return False
    def do_parse_raw_file(self, path):
        f = open(path, "r+b")
        img = f.read()
        f.close()
        return img
    def status_check(self):
        type_len = self.read_all(16)
        if not type_len:
            print "error,unknow type file"
        self.system_status = self.read_all(int(type_len))
from RCPCom.TcpServer import TcpServer
from RCPCom.RCPClient import RCPClient
from RCPCom.RCPInputQueueManager import InputQueueManager
from RCPCom.RCPOutputQueueManager import OutputQueueManager
from RCPCom.RCPDatagramAnalyser import RCPDatagramAnalyser
from RCPCom.RCPDecodingTask import RCPDecodingTask
from RCPCom.RCPEncodingTask import RCPEncodingTask
import sys
class RCPComStack():
    def __init__(self, context):
        self.context = context
        self.inputQueueManager = InputQueueManager()
        self.outputQueueManager = OutputQueueManager()
        self.datagramAnalyser = RCPDatagramAnalyser(self, self.context)
        self.serv = TcpServer(self.inputQueueManager, 10704)
        self.serv.create_server()
        self.decodingTask = RCPDecodingTask(self.inputQueueManager, self.context, self.datagramAnalyser)
        self.encodingTask = RCPEncodingTask(self.context, self.outputQueueManager)
        self.clientList = list()
    def connectera(self, ip, port):
        client = RCPClient(self.outputQueueManager)
        client.connectera(ip, port)
        self.clientList.append(client)
    def launch_transmission_task_by_addr(self, addr):
        for client in self.clientList:
            if client.get_addr() == addr:
                client.launch()
    def close_session(self):
        self.context.close_system()
        self.serv.terminate_server()
        self.decodingTask.stop()
        self.encodingTask.stop()
        sys.exit(0)
class RCPDatagram:
    def __init__(self, msg):
        self.data_type = 0
        self.origin_id = 0
        self.target_id = 0
        self.timestamps = 123456
        self.dlc = 4
        self.body = "
        self.decode(msg)
    def get_data_type(self):
        return self.data_type
    def set_data_type(self, data_type):
        self.data_type = data_type
    def get_target_id(self):
        return self.target_id
    def set_target_id(self, target_id):
        self.target_id = target_id
    def get_origine_id(self):
        return self.origin_id
    def set_origine_id(self, origin_id):
        self.origin_id = origin_id
    def get_time_stamps(self):
        return self.timestamps
    def set_time_stamps(self, time_stampes):
        self.timestamps = time_stampes
    def get_dlc(self):
        return self.dlc
    def set_dlc(self, dlc):
        self.dlc = dlc
    def get_itc_datagram_body(self):
        return self.body
    def decode(self, byte_array):
        self.data_type = ord(byte_array[0]) + ord(byte_array[1])*256
        self.origin_id = ord(byte_array[2])
        self.target_id = ord(byte_array[3])
        self.timestamps = ord(byte_array[4]) + ord(byte_array[5])*256 +
ord(byte_array[6])*256*256 + ord(byte_array[7])*256*256*256
        self.dlc = ord(byte_array[8]) + ord(byte_array[9])*256
        self.body = byte_array[10:1024]
    def encode(self):
        timestamps_msb = self.timestamps/(2**16)
        timestamps_lsb = self.timestamps % (2**16)
        msg = chr(self.data_type % 256) + chr(self.data_type/256) \
              + chr(self.origin_id) + chr(self.target_id)\
              + chr(timestamps_lsb % 256) + chr(timestamps_lsb/256) \
              + chr(timestamps_msb % 256) + chr(timestamps_msb/256) \
              + chr(self.dlc % 256) + chr(self.dlc/256)
        msg_len = len(msg)
        for x in range(msg_len, 1024):
            msg[x] = self.body[x - msg_len]
        return msg
from RCPContext.RCPContext import RCPContext
import RCPDatagram
from MotorMsg import MotorMsg
from InjectionMsg import InjectionMsg
class RCPDatagramAnalyser:
    def __init__(self, parent, context):
        self.parent = parent
        self.context = context
        self.switcher = {
            0: "HelloMsg",
            1: "HandShakeMsg",
            2: "HandShakeCommitMsg",
            3: "MotorMsg",
            4: "CTImage",
            9: "InjectionMsg",
           10: "CloseSessionMsg"
        }
        self.switcher_instruction = {
            0: "catheterMoveInstruction",
            1: "guidewireProgressInstruction",
            2: "guidewireRotateInstruction",
            3: "contrastMediaPushInstruction",
            4: "retractInstruction"
        }
    def analyse(self, cpt, datagram):
            if self.switcher[datagram.get_data_type()] == "HelloMsg":
                self.decode_hello_message(datagram)
            elif self.switcher[datagram.get_data_type()] == "HandShakeMsg":
                pass
            elif self.switcher[datagram.get_data_type()] == "HandShakeCommitMsg":
                self.decode_handshake_commit_message(datagram)
            elif self.switcher[datagram.get_data_type()] == "MotorMsg":
                self.decode_motor_message(datagram)
            elif self.switcher[datagram.get_data_type()] == "CTImage":
                pass
            elif self.switcher[datagram.get_data_type()] == "CloseSessionMsg":
                self.decode_close_session_message(datagram)
            elif self.switcher[datagram.get_data_type()] == "InjectionMsg":
                self.decode_injection_message(datagram)
    def decode_injection_message(self, datagram):
        datagram_body = datagram.get_itc_datagram_body()
        injection_msg = InjectionMsg(datagram)
        self.context.append_new_injection_msg(injection_msg)
    def decode_close_session_message(self, datagram):
        datagram_body = datagram.get_itc_datagram_body()
        self.parent.close_session()
    def decode_hello_message(self, datagram):
        x = 1
    def decode_motor_message(self, datagram):
        motor_msg = MotorMsg(datagram)
        if self.switcher_instruction[motor_msg.motor_type] == "catheterMoveInstruction":
            self.context.append_new_catheter_move_message(motor_msg)
        elif self.switcher_instruction[motor_msg.motor_type] ==
"guidewireProgressInstruction":
            self.context.append_new_guidewire_progress_move_message(motor_msg)
        elif self.switcher_instruction[motor_msg.motor_type] == "guidewireRotateInstruction":
            self.context.append_new_guidewire_rotate_move_message(motor_msg)
        elif self.switcher_instruction[motor_msg.motor_type] ==
"contrastMediaPushInstruction":
            self.context.append_new_contrast_media_push_move_message(motor_msg)
        elif self.switcher_instruction[motor_msg.motor_type] == "retractInstruction":
            self.context.append_latest_retract_message(motor_msg)
    def decode_handshake_commit_message(self, datagram):
        datagram_body = datagram.get_itc_datagram_body()
addr = str(ord(datagram_body[0])) + '.' + str(ord(datagram_body[1])) + '.' +
str(ord(datagram_body[2])) + '.' + str(ord(datagram_body[3]))
        self.parent.launch_transmission_task_by_addr(addr)
import threading
import time
class RCPDecodingTask:
    def __init__(self, input_queue_manager, _context, _datagram_analyser):
        self.inputQueueManager = input_queue_manager
        self.context = _context
        self.datagramAnalyser = _datagram_analyser
        self.flag = True
        self.receptionTask = threading.Thread(None, self.decodage)
        self.receptionTask.start()
    def stop(self):
        self.flag = False
    def decodage(self):
        while self.flag:
            if self.inputQueueManager.get_length() > 0:
                for cpt in range(0, self.inputQueueManager.get_length()):
                    if
self.inputQueueManager.get_data_array_count_from_input_queue(cpt) > 0:
                        ret =
self.inputQueueManager.get_data_array_from_input_queue(cpt)
                        self.datagramAnalyser.analyse(cpt, ret)
            time.sleep(0.03)
import io
import os
import socket
import struct
import mmap
import threading
import time
class RCPEncodingTask:
    def __init__(self, context, output_queue_manager):
        self.context = context
        self.output_queue_manager = output_queue_manager
        self.flag = True
        self.encodingThread = threading.Thread(None, self.decodage)
    def stop(self):
        self.flag = False
    def decodage(self):
        while self.flag:
            if self.output_queue_manager.get_length() > 0:
                for cpt in range(0, self.output_queue_manager.get_length()):
                    if
self.context.get_latest_guidewire_moving_distance_sequence_length()>0:
                msg = self.context.fetch_latest_guidewire_moving_distance_msg()
                self.output_queue_manager.add_datagram_by_id(cpt, msg)
            time.sleep(0.05)
import threading
class InputQueue:
    def __init__(self):
        self.inputQueueLock = threading.Lock()
        self.inputQueue = list()
    def append(self, datagram):
        self.inputQueue.append(datagram)
    def get_latest_array(self):
        self.inputQueueLock.acquire()
        if len(self.inputQueue) > 0:
            ret = self.inputQueue.pop(0)
        self.inputQueueLock.release()
        return ret
    def get_length(self):
        self.inputQueueLock.acquire()
        length = len(self.inputQueue)
        self.inputQueueLock.release()
        return length
from RCPCom.TcpServer import TcpServer
from RCPCom.RCPClient import RCPClient
import threading
class InputQueueManager():
    def __init__(self):
        self.rcpInputQueueManager = list()
        self.rcpInputQueueManagerLock = threading.Lock()
    def add_rcp_input_queue(self, input_queue):
        self.rcpInputQueueManager.append(input_queue)
    def add_datagram_by_id(self, id, datagram):
        self.rcpInputQueueManager[id].append(datagram)
    def get_length(self):
        self.rcpInputQueueManagerLock.acquire()
        ret = len(self.rcpInputQueueManager)
        self.rcpInputQueueManagerLock.release()
        return ret
    def get_data_array_count_from_input_queue(self, cpt):
        self.rcpInputQueueManagerLock.acquire()
        ret = self.rcpInputQueueManager[cpt].get_length()
        self.rcpInputQueueManagerLock.release()
        return ret
    def get_data_array_from_input_queue(self, cpt):
        self.rcpInputQueueManagerLock.acquire()
        ret = self.rcpInputQueueManager[cpt].get_latest_array()
        self.rcpInputQueueManagerLock.release()
        return ret
from RCPCom.TcpServer import TcpServer
import threading
class OutputQueue():
    def __init__(self):
    self.outputQueueLock = threading.Lock()
        self.outputQueue = []
    def append(self, datagram):
        self.outputQueue.append(datagram)
    def get_latest_array(self):
        self.outputQueueLock.acquire()
        if len(self.outputQueue) > 0:
            ret = self.outputQueue.pop(0)
        self.outputQueueLock.release()
        return ret
    def get_length(self):
        self.outputQueueLock.acquire()
        length = len(self.outputQueue)
        self.outputQueueLock.release()
        return length
import threading
class OutputQueueManager():
    def __init__(self):
        self.rcpOutputQueueManager = list()
        self.rcpOutputQueueManagerLock = threading.Lock()
    def add_rcp_output_queue(self, output_queue):
        self.rcpOutputQueueManagerLock.acquire()
        self.rcpOutputQueueManager.append(output_queue)
        self.rcpOutputQueueManagerLock.release()
def add_datagram_by_id(self, id, datagram):
        self.rcpOutputQueueManagerLock.acquire()
        self.rcpOutputQueueManager[id].append(datagram)
        self.rcpOutputQueueManagerLock.release()
def get_length(self):
        self.rcpOutputQueueManagerLock.acquire()
        ret = len(self.rcpOutputQueueManager)
        self.rcpOutputQueueManagerLock.release()
        return ret
    def get_data_array_count_from_output_queue(self, cpt):
        self.rcpOutputQueueManagerLock.acquire()
        ret = self.rcpOutputQueueManager[cpt].get_length()
        self.rcpOutputQueueManagerLock.release()
        return ret
    def get_data_array_from_output_queue(self, cpt):
        self.rcpOutputQueueManagerLock.acquire()
        ret = self.rcpOutputQueueManager[cpt].get_latest_array()
        self.rcpOutputQueueManagerLock.release()
        return ret
import io
import socket
import threading
from IncomingClient import Client
import os
import threading as td
from RCPInputQueue import InputQueue
class TcpServer:
    def __init__(self, input_queue_manager, port):
        self.inputQueueManager = input_queue_manager
        self.port = port
        self.userNum = 0
        self.server_socket = None
        self.flag = True
        self.listeningTask = threading.Thread(None, self.listening)
        self.clientList = list()
    def create_server(self):
        self.server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.server_socket.bind(('0.0.0.0', self.port))
        self.server_socket.listen(0)
        self.listeningTask.start()
    def terminate_server(self):
        print "socket server close"
        self.flag = False
        self.server_socket.close()
    def listening(self):
        while self.flag:
            print 'waiting for the client:', self.userNum
            connection, address = self.server_socket.accept()
            print 'incoming connection...', address
            input_queue = InputQueue()
            self.inputQueueManager.add_rcp_input_queue(input_queue)
            client = Client(connection, address, self.userNum, self.inputQueueManager)
            client.enable()
            self.clientList.append(client)
            self.userNum += 1
    def set_current_state(self, current_state):
        for client in self.clientList:
            client.set_current_state(current_state)
    def launch(self):
        self.create_server(
    def close(self):
        self.flag = False
import threading
class RCPContext:
    def __init__(self):
        self.inputLock = threading.Lock()
        self.outputLock = threading.Lock()
        self.catheterMoveInstructionSequence = []
        self.guidewireProgressInstructionSequence = []
        self.guidewireRotateInstructionSequence = []
self.guidewireMovingDistance = []
        self.contrastMediaPushInstructionSequence = []
self.injectionCommandSequence = []
        self.retractInstructionSequence = []
`       self.closeSessionSequence = []

        self.systemStatus = True
    def append_close_session_msg(self, close_session_msg):
        self.closeSessionSequence.append(close_session_msg)
    def fetch_close_session_msg(self):
        self.inputLock.acquire()
        length = len(self.closeSessionSequence)
        ret = self.closeSessionSequence.pop(length-1)
        self.inputLock.release()
        return ret
    def get_close_session_sequence_length(self):
        self.inputLock.acquire()
        length = len(self.closeSessionSequence)
        self.inputLock.release()
        return length
    def append_new_injection_msg(self, msg):
        self.inputLock.acquire()
        self.injectionCommandSequence.append(msg)
        self.inputLock.release()
    def fetch_latest_injection_msg_msg(self):
        self.inputLock.acquire()
        length = len(self.injectionCommandSequence)
        ret = self.injectionCommandSequence.pop(length-1)
        self.inputLock.release()
        return ret
    def get_injection_command_sequence_length(self):
        self.inputLock.acquire()
        length = len(self.injectionCommandSequence)
        self.inputLock.release()
        return length
 def close_system(self):
        self.systemStatus = False
        self.catheterMoveInstructionSequence = []
        self.guidewireProgressInstructionSequence = []
        self.guidewireRotateInstructionSequence = []
        self.contrastMediaPushInstructionSequence = []
        self.retractInstructionSequence = []
        self.guidewireMovingDistance = []
        self.closeSessionSequence = []
    def open_system(self):
        self.systemStatus = True
    def get_system_status(self):
        return self.systemStatus
    def clear(self):
        self.catheterMoveInstructionSequence = []
        self.guidewireProgressInstructionSequence = []
        self.guidewireRotateInstructionSequence = []
        self.contrastMediaPushInstructionSequence = []
        self.retractInstructionSequence = []
        self.guidewireMovingDistance = []
        self.closeSessionSequence = []
    def set_distance(self, dis):
        self.guidewireMovingDistance.append(dis)
    def fetch_latest_guidewire_moving_distance(self):
        self.outputLock.acquire()
        length = len(self.guidewireMovingDistance)
        ret = self.guidewireMovingDistance[length-1]
        self.outputLock.release()
        return ret
     def connectera(self, addr, port):
        print "connect server", addr, port
        self.addr = addr
        self.connection = socket.socket()
        self.connection.connect((addr, port))
        for i in range(3):
            self.send_handshake_message()
    def launch_trasmission_task(self):
        print "connected... start real time communication task"
        self.launching = True
        self.rtTask.start()
    def fermeture(self):
        self.connection.\
            close()
    def task(self):
        if len(self.msg_list) > 0:
            self.connection.sendall(self.msg_list.pop(0))
    def execute_rt_task(self):
        while self.launching:
            self.msg_producer()
            time.sleep(0.1)
        self.fermeture()
    def read_all(self, count):
        buf = b''
        while count:
            receiving_buffer = self.clientSocket.recv(count)
            if not receiving_buffer:
        return None
   def fetch_latest_guidewire_moving_distance_msg(self):
        self.outputLock.acquire()
        length = len(self.guidewireMovingDistance)
        ret = self.guidewireMovingDistance.pop(length-1)
        self.outputLock.release()
        return ret
    def get_latest_guidewire_moving_distance_sequence_length(self):
        self.outputLock.acquire()
        length = len(self.guidewireMovingDistance)
        self.outputLock.release()
        return length
    def append_new_catheter_move_message(self, msg):
        self.inputLock.acquire()
        self.catheterMoveInstructionSequence.append(msg)
        self.inputLock.release()
    def fetch_latest_catheter_move_msg(self):
        self.inputLock.acquire()
        length = len(self.catheterMoveInstructionSequence)
        ret = self.catheterMoveInstructionSequence.pop(length-1)
        self.inputLock.release()
        return ret
    def get_catheter_move_instruction_sequence_length(self):
        self.inputLock.acquire()
        length = len(self.catheterMoveInstructionSequence)
        self.inputLock.release()
        return length
    def append_new_guidewire_progress_move_message(self, msg):
        self.inputLock.acquire()
        self.guidewireProgressInstructionSequence.append(msg)
        self.inputLock.release()
    def fetch_latest_guidewire_progress_move_msg(self):
        self.inputLock.acquire()
        length = len(self.guidewireProgressInstructionSequence)
        ret = self.guidewireProgressInstructionSequence.pop(length-1)
        self.inputLock.release()
        return ret
    def get_guidewire_progress_instruction_sequence_length(self):
        self.inputLock.acquire()
        length = len(self.guidewireProgressInstructionSequence)
        self.inputLock.release()
        return length

    def append_new_guidewire_rotate_move_message(self, msg):
        self.inputLock.acquire()
        self.guidewireRotateInstructionSequence.append(msg)
        self.inputLock.release()
    def fetch_latest_guidewire_rotate_move_msg(self):
        self.inputLock.acquire()
        length = len(self.guidewireRotateInstructionSequence)
        ret = self.guidewireRotateInstructionSequence.pop(length-1)
        self.inputLock.release()
        return ret
    def get_guidewire_rotate_instruction_sequence_length(self):
        self.inputLock.acquire()
        length = len(self.guidewireRotateInstructionSequence)
        self.inputLock.release()
        return length
    def append_new_contrast_media_push_move_message(self, msg):
        self.inputLock.acquire()
        self.contrastMediaPushInstructionSequence.append(msg)
        self.inputLock.release()
    def fetch_latest_contrast_media_push_move_msg(self):
        self.inputLock.acquire()
        length = len(self.contrastMediaPushInstructionSequence)
        ret = self.contrastMediaPushInstructionSequence.pop(length-1)
        self.inputLock.release()
        return ret
    def get_contrast_media_push_instruction_sequence_length(self):
        self.inputLock.acquire()
        length = len(self.contrastMediaPushInstructionSequence)
        self.inputLock.release()
        return length
    def append_latest_retract_message(self, msg):
        self.inputLock.acquire()
        self.retractInstructionSequence.append(msg)
        self.inputLock.release()
    def fetch_latest_retract_msg(self):
        self.inputLock.acquire()
        length = len(self.retractInstructionSequence)
        ret = self.retractInstructionSequence.pop(length-1)
        self.inputLock.release()
        return ret
    def get_retract_instruction_sequence_length(self):
        self.inputLock.acquire()
        length = len(self.retractInstructionSequence)
        self.inputLock.release()
        return length
import threading
import time
import sys
from RCPContext.RCPContext import RCPContext
from OrientalMotor import OrientalMotor
from Gripper import Gripper
from MaxonMotor import MaxonMotor
from InfraredReflectiveSensor import InfraredReflectiveSensor
from EmergencySwitch import EmergencySwitch
class Dispatcher(object):
    def __init__(self, context, local_mode=0):
        self.context = context
        self.flag = True
        self.cptt = 0
        self.global_state = 0
        self.needToRetract = False
        self.draw_back_guidewire_curcuit_flag = True
        self.number_of_cycles = 0
        self.guidewireProgressMotor = OrientalMotor(20, 21, True)
        self.guidewireRotateMotor = OrientalMotor(19, 26, True)
        self.catheterMotor = OrientalMotor(17, 27, True)
        self.angioMotor = OrientalMotor(23, 24, False)
        self.gripperFront = Gripper(7)
        self.gripperBack = Gripper(8)
        self.infraredReflectiveSensor = InfraredReflectiveSensor()
        self.switch = EmergencySwitch()
        self.emSwitch = 1
        self.lastSwitch = 0
        self.em_count = 0
        self.speedProgress = 1000
        self.speedRotate = 60
        self.speedCatheter =10
        self.rotateTime = 180/self.speedRotate
        self.pos_speed = 5
        self.position_cgf = 2
        self.position_cgb = -100
    if local_mode == 0:
            self.dispatchTask = threading.Thread(None, self.do_parse_commandes_in_context)
            self.dispatchTask.start()
    def set_global_state(self, state):
        self.global_state = state
    def do_parse_commandes_in_context(self):
        while self.flag:
            if not self.context.get_system_status():
                self.guidewireRotateMotor.close_device()
                self.guidewireProgressMotor.close_device()
                self.catheterMotor.close_device()
                self.angioMotor.close_device()
                sys.exit()
                self.flag = False
                print "system terminated"
             else:
                self.emSwitch = self.switch.read_current_state()
                if self.emSwitch == 1:

                    time.sleep(0.02)
                    self.guidewireRotateMotor.standby()
                    self.guidewireProgressMotor.standby()
                    self.catheterMotor.standby()
                    self.angioMotor.standby()
                    self.lastSwitch = 1
                elif self.emSwitch == 0 and self.lastSwitch == 1:
                    self.guidewireRotateMotor.enable()
                    self.guidewireProgressMotor.enable()
                    self.catheterMotor.enable()
                    self.angioMotor.enable()
                    self.lastSwitch = 0
                    self.decode()
                elif self.emSwitch == 0 and self.lastSwitch == 0:
                    #print 'start', self.emSwitch
                    self.decode()
                    time.sleep(0.03)
    def decode(self):
        if self.context.get_catheter_move_instruction_sequence_length() > 0:
            msg = self.context.fetch_latest_catheter_move_msg()
            if self.draw_back_guidewire_curcuit_flag == False:
                return
            if msg.get_motor_orientation() == 0:
                    self.catheterMotor.set_speed(msg.get_motor_speed()/10.0)
                    return
            elif msg.get_motor_orientation() == 1:
            self.catheterMotor.set_speed(-msg.get_motor_speed()/10.0)
            return
        if not self.needToRetract:
            if self.context.get_guidewire_progress_instruction_sequence_length() > 0:
                self.set_global_state(self.infraredReflectiveSensor.read_current_state())
                if self.global_state == 0:
                    msg = self.context.fetch_latest_guidewire_progress_move_msg()
                if self.draw_back_guidewire_curcuit_flag == False:
                    return
                  if msg.get_motor_orientation() == 0 and abs(msg.get_motor_speed()) <
40*2*60:
                    self.guidewireProgressMotor.set_speed(-msg.get_motor_speed())
                    self.cptt = 0
                elif msg.get_motor_orientation() == 1 and abs(msg.get_motor_speed()) <
40*2*60:
                    self.guidewireProgressMotor.set_speed(msg.get_motor_speed())
            else:
                self.guidewireProgressMotor.set_speed(0)
            elif self.global_state == 2:
                self.guidewireProgressMotor.set_speed(0)
                self.needToRetract = True
                retractTask = threading.Thread(None, self.push_guidewire_back)
                retractTask.start()
            elif self.global_state == 1:
                print "hehe", self.global_guidewire_distance
                self.guidewireProgressMotor.set_speed(self.speedProgress)
            elif self.global_state == 3:
                self.guidewireProgressMotor.set_speed(0)
            if self.context.get_guidewire_rotate_instruction_sequence_length() > 0:
                msg = self.context.fetch_latest_guidewire_rotate_move_msg()
                speed = msg.get_motor_speed()
                position = (msg.get_motor_position()*4000)/360
                if self.draw_back_guidewire_curcuit_flag == False:
                    return
                if msg.get_motor_orientation() == 0:
                    self.guidewireRotateMotor.set_speed(speed)
                    pass
                elif msg.get_motor_orientation() == 1:
                    self.guidewireRotateMotor.set_speed(-speed)
                    pass
        if self.context.get_contrast_media_push_instruction_sequence_length() > 0:
            msg = self.context.fetch_latest_contrast_media_push_move_msg()
            ret = msg.get_motor_speed()
        if self.draw_back_guidewire_curcuit_flag == False:
            return
        if msg.get_motor_orientation() == 0:
            self.angioMotor.set_speed(-ret)
        elif msg.get_motor_orientation() == 1:
            self.angioMotor.set_speed(ret)
        if self.context.get_retract_instruction_sequence_length() > 0:
            if self.draw_back_guidewire_curcuit_flag == False:
                return
                self.draw_back_guidewire_curcuit()
        if self.context.get_injection_command_sequence_length() > 0:
             msg = self.context.fetch_latest_injection_msg_msg()
        if msg.get_volume() < 0:
            self.angioMotor.set_pos_speed(msg.get_speed())
            self.angioMotor.set_position(msg.get_volume()/4.5)
            self.angioMotor.pull_contrast_media()
        elif msg.get_volume() <= 30:
            self.angioMotor.set_pos_speed(msg.get_speed())
            self.angioMotor.set_position(msg.get_volume()/4.5)
            self.angioMotor.push_contrast_media()
def push_contrast_agent(self):
    self.angioMotor.set_pos_speed(self.pos_speed)
    self.angioMotor.set_position(self.position_cgf/4.5)
    self.angioMotor.push_contrast_media()
def pull_contrast_agent(self):
    self.angioMotor.set_pos_speed(self.pos_speed)
    self.angioMotor.set_position(self.position_cgb/4.5)
    self.angioMotor.pull_contrast_media()
def push_guidewire_back(self):
    self.draw_back_guidewire_curcuit_flag == False
    self.gripperFront.gripper_chuck_fasten()
    self.gripperBack.gripper_chuck_fasten()
    time.sleep(1)
    self.guidewireRotateMotor.set_speed(-self.speedRotate) # +/loosen
    time.sleep(self.rotateTime)
    self.guidewireRotateMotor.set_speed(0)
    self.guidewireProgressMotor.set_speed(-self.speedProgress)
    self.global_state = self.infraredReflectiveSensor.read_current_state()
    while self.global_state != 1:
        self.global_state = self.infraredReflectiveSensor.read_current_state()
        if self.global_state == 4:
            self.global_state = self.infraredReflectiveSensor.read_current_state()
            continue
        time.sleep(0.5)
        self.global_state = self.infraredReflectiveSensor.read_current_state()
    print "retracting", self.global_state
    print "back limitation arrived"
```

```
        self.guidewireProgressMotor.set_speed(0)
        self.guidewireRotateMotor.set_speed(self.speedRotate)
        time.sleep(self.rotateTime)
        self.guidewireRotateMotor.set_speed(0)
        self.gripperFront.gripper_chuck_loosen()
        self.gripperBack.gripper_chuck_loosen()
        self.draw_back_guidewire_curcuit_flag == True
self.needToRetract = False
   def push_guidewire_advance(self):
        self.guidewireProgressMotor.set_speed(self.speedProgress)
        self.global_state = self.infraredReflectiveSensor.read_current_state()
        while self.global_state !=2:
            time.sleep(0.5)
            self.global_state = self.infraredReflectiveSensor.read_current_state()
        self.guidewireProgressMotor.set_speed(0)
    def multitime_push_guidewire(self):
        self.define_number_of_cycles()
        for i in range (0,self.number_of_cycles):
            self.push_guidewire_advance()
            self.push_guidewire_back()
            print(i)
    def draw_guidewire_back(self):
        self.guidewireRotateMotor.set_speed(self.speedRotate)
        time.sleep(self.rotateTime)
        self.guidewireRotateMotor.set_speed(0)
        self.gripperBack.gripper_chuck_loosen()
        time.sleep(1)
        self.guidewireProgressMotor.set_speed(-self.speedProgress)
        self.global_state = self.infraredReflectiveSensor.read_current_state()
        while self.global_state != 1:
            time.sleep(0.5)
            self.global_state = self.infraredReflectiveSensor.read_current_state()
        self.guidewireProgressMotor.set_speed(0)
    def chuck_loosen(self):
        self.gripperBack.gripper_chuck_fasten()
        time.sleep(1)
        self.guidewireRotateMotor.set_speed(-self.speedRotate)
        time.sleep(self.rotateTime)
        self.guidewirRotateMotor.set_speed(0)
    def chuck_fasten(self):
        self.gripperBack.gripper_chuck_fasten()
        time.sleep(1)
        self.guidewireRotateMotor.set_speed(self.speedRotate)
        time.sleep(self.rotateTime)
        self.guidewireRotateMotor.set_speed(0)
    def draw_guidewire_advance(self):
        self.gripperFront.gripper_chuck_loosen()
        self.gripperBack.gripper_chuck_loosen()
        time.sleep(1)
        self.gripperFront.gripper_chuck_fasten()
    self.gripperBack.gripper_chuck_fasten()
    time.sleep(1)
    self.guidewireRotateMotor.set_speed(-self.speedRotate)
    time.sleep(self.rotateTime)
    self.guidewireRotateMotor.set_speed(0)
    self.guidewireProgressMotor.set_speed(self.speedProgress)
    self.global_state = self.infraredReflectiveSensor.read_current_state()
    while self.global_state !=2:
        time.sleep(0.5)
        self.global_state = self.infraredReflectiveSensor.read_current_state()
    self.guidewireProgressMotor.set_speed(0)
    time.sleep(1)
def multitime_draw_back_guidewire(self):
    self.define_number_of_cycles()
    for i in range (0,self.number_of_cycles):
        self.draw_guidewire_advance()
        self.draw_guidewire_back()
        print(i)
def automatic_procedure(self):
    self.angioMotor.set_pos_speed(4)
    self.angioMotor.set_position(10)
    self.angioMotor.push_contrast_media()
    print "angiographing finish"
    time.sleep(5)
    self.multitime_push_guidewire()
def push_and_pull(self):
    self.multitime_push_guidewire()
    self.multitime_draw_back_guidewire()
def loosen(self):
    self.gripperBack.gripper_chuck_fasten()
    time.sleep(1)
    self.gripperBack.gripper_chuck_loosen()
    time.sleep(1)
def catheter_advance(self):
    self.gripperFront.gripper_chuck_loosen()
    self.gripperBack.gripper_chuck_loosen()
    self.draw_back_guidewire_curcuit_flag == True
    self.needToRetract = False
    self.guidewireProgressMotor.set_speed(self.speedProgress)
    self.catheterMotor.set_speed(self.speedCatheter)
    self.global_state = self.infraredReflectiveSensor.read_current_state()
    while self.global_state !=2:
        time.sleep(0.5)
        self.global_state = self.infraredReflectiveSensor.read_current_state()
    self.guidewireProgressMotor.set_speed(0)
    self.catheterMotor.set_speed(0)
    self.gripperFront.gripper_chuck_fasten()
    self.gripperBack.gripper_chuck_fasten()
    time.sleep(1)
    self.guidewireRotateMotor.set_speed(-self.speedRotate) # +/loosen
```

```
        time.sleep(self.rotateTime)
        self.guidewireRotateMotor.set_speed(0)
        self.guidewireProgressMotor.set_speed(-self.speedProgress)
        self.global_state = self.infraredReflectiveSensor.read_current_state()
        while self.global_state != 1:
            time.sleep(0.5)
            self.global_state = self.infraredReflectiveSensor.read_current_state()
            print "retracting", self.global_state
            print "back limitation arrived"
        self.guidewireProgressMotor.set_speed(0)
        self.guidewireRotateMotor.set_speed(self.speedRotate)
        time.sleep(self.rotateTime)
        self.guidewireRotateMotor.set_speed(0)
    def multitime_catheter_advance(self):
        self.define_number_of_cycles()
        for i in range (0,self.number_of_cycles):
            self.catheter_advance()
    def test(self):
        self.gripperBack.gripper_chuck_fasten()
    def catheter_back(self):
        self.define_number_of_cycles()
        for i in range (0,self.number_of_cycles):
            self.draw_guidewire_back()
            self.catheterMotor.set_speed(self.speedCatheter)
            print(i)
    def initialization(self):
self.guidewireProgressMotor.set_speed(-self.speedProgress)
self.global_state = self.infraredReflectiveSensor.read_current_state()
        while self.global_state != 1:
            time.sleep(0.5)
            self.global_state = self.infraredReflectiveSensor.read_current_state()
            print "retracting", self.global_state
            print "back limitation arrived"
        self.guidewireProgressMotor.set_speed(0)
        self.guidewireRotateMotor.set_speed(self.speedRotate)
        time.sleep(self.rotateTime)
        self.guidewireRotateMotor.set_speed(0)
        self.gripperFront.gripper_chuck_loosen()
        self.gripperBack.gripper_chuck_loosen()
        self.draw_back_guidewire_curcuit_flag == True
        self.needToRetract = False
    def catheter(self):
        self.catheterMotor.set_speed(self.speedCatheter)
    def define_number_of_cycles(self):
        self.number_of_cycles = input("please input the number of cycles")
import RPi.GPIO as GPIO
import time
import threading
import random
class EmergencySwitch(object):
    def __init__(self):
        self.switch = 5
        GPIO.setmode(GPIO.BCM)
        GPIO.setwarnings(False)
        GPIO.setup(self.switch, GPIO.IN)
    def read_current_state(self):
        sw = GPIO.input(self.switch)
        return sw
mport RPi.GPIO as GPIO
import time
import threading
import random
class InfraredReflectiveSensor(object):
    def __init__(self):
        self.switch = 22
        self.flag = True
        GPIO.setmode(GPIO.BCM)
        GPIO.setwarnings(False)
        GPIO.setup(self.switch, GPIO
        GPIO.setup(self.doutFront, GPIO.IN)
    def read_current_state(self):
        back = GPIO.input(self.doutBack)
        front = GPIO.input(self.doutFront)
        if back == 0 and front == 1:
            return 1
        if back == 1 and front == 0:
            return 2
        if back == 0 and front == 0:
            return 3
        return 0
    def read(self):
        cpt = 0
        while self.flag:
            self.read_current_state()
            time.sleep(0.5)
import RPi.GPIO as GPIO
import time
class Gripper(object):
    def __init__(self, io):
        GPIO.setmode(GPIO.BCM)
        GPIO.setwarnings(False)
        self.flag = True
        self.count = 0
        self.io = io
        GPIO.setup(self.io, GPIO.OUT, initial=GPIO.LOW)
    def gripper_chuck_fasten(self):
        GPIO.output(self.io, True)
    def gripper_chuck_loosen(self):
        GPIO.output(self.io, False)
mport RPi.GPIO as GPIO
```

```
import time
import threading
import random
from EmergencySwitch import EmergencySwitch
class InfraredReflectiveSensor(object):
    def __init__(self):
        self.doutBack = 2
        self.doutFront = 3
        self.flag = True
        GPIO.setmode(GPIO.BCM)
        GPIO.setwarnings(False)
        GPIO.setup(self.doutBack, GPIO.IN)
        GPIO.setup(self.doutFront, GPIO.IN)
        self.switch = EmergencySwitch()
    def read_current_state(self):
        back = GPIO.input(self.doutBack)
        front = GPIO.input(self.doutFront)
        emSwitch = self.switch.read_current_state()
        if emSwitch == 1:
            return 4
        else:
            if back == 0 and front == 1:
                #print 'start move'
                return 1
            if back == 1 and front == 0:
                #print 'start retract'
                return 2
             if back == 0 and front == 0:
                return 3
        return 0
    def read(self):
        cpt = 0
        while self.flag:
            self.read_current_state()
            time.sleep(0.5)
import RPi.GPIO as GPIO
import time
import threading
class OrientalMotor(object):
    def __init__(self, push_io, pull_io, mode_flag):
        self.orientalMotorPushLock = threading.Lock()
        self.orientalMotorPullLock = threading.Lock()
        GPIO.setmode(GPIO.BCM)
        GPIO.setwarnings(False)
        self.flag = True
        self.pos_flag = True
        self.speedFlag = 0
        self.count = 0
        self.pushIO = push_io
        self.pullIO = pull_io
        GPIO.setup(self.pushIO, GPIO.OUT, initial=GPIO.HIGH)
        GPIO.setup(self.pullIO, GPIO.OUT, initial=GPIO.HIGH)
        self.mode = mode_flag
        self.speed = 0
        self.pos_motor_flag = 1
        self.re_vol_pos = 1        # 3.4
        self.position = 0
        self.re_volsp_possp = 204.0
        self.pos_speed = 60.0
        self.pos_count = 0
        self.mv_enable = True
if self.mode:
            self.moveTask = threading.Thread(None, self.continuous_move)
            self.moveTask.start()
    def open_device(self):
        self.flag = True
    def close_device(self):
        self.flag = False
    def close_position_device(self):
        self.pos_flag = False
    def set_speed(self, speed):
        if speed > 0:
            self.speedFlag = 1
        elif speed < 0:
            self.speedFlag = 2
        elif speed == 0:
        self.speedFlag = 0
        self.speed = abs(speed)
    def standby(self):
        self.mv_enable = False
    def enable(self):
        self.mv_enable = True
    def continuous_move(self):
        while self.flag:
            if self.mv_enable:
                if self.speedFlag == 0:
            time.sleep(0.1)
                if self.speedFlag == 1:
                    self.push()
                if self.speedFlag == 2:
                    self.pull()
            else:
                time.sleep(0.5)
    def rtz(self):
        GPIO.output(self.pushIO, True)
        GPIO.output(self.pullIO, True)
    def push(self):
        self.orientalMotorPushLock.acquire()
        if self.speed == 0:
            interval = 0
        else:
            interval = 0.0005*60/self.speed
        GPIO.output(self.pushIO, False)
        time.sleep(interval)
        GPIO.output(self.pushIO, True)
        time.sleep(interval)
        self.count += 1
        self.orientalMotorPushLock.release()
    def pull(self):
        self.orientalMotorPullLock.acquire()
        if self.speed == 0:
            interval = 0
        else:
            interval = 0.0005*60/self.speed
        GPIO.output(self.pullIO, False)
        time.sleep(interval)
        GPIO.output(self.pullIO, True)
        time.sleep(interval)
        self.count += 1
        self.orientalMotorPullLock.release()
    def set_position(self, volume):
        self.position = volume*self.re_vol_pos*2
    def set_pos_speed(self, vol_speed):
        self.pos_speed = vol_speed*self.re_volsp_possp
    def continuous_move_position(self):
        while self.pos_flag:
            if self.position > 0:
                self.position_push()
        time.sleep(self.get_position_sleep_time())
            elif self.position < 0:
                self.position_pull()
        time.sleep(self.get_position_sleep_time())
        elif self.position == 0:
        time.sleep(0.001)
    def push_contrast_media(self):
        self.position_push()
        self.pos_count+= self.position/self.re_vol_pos
        time.sleep(0.001)
    def pull_contrast_media(self):
        self.position_pull()
    def pull_back(self):
        self.set_position(self.pos_count/2)
        self.set_pos_speed(self.pos_speed/self.re_volsp_possp/2)
        self.position_pull()
        self.stop()
    def stop(self):
        self.set_position(0)
        self.set_pos_speed(0)
        self.position_pull()
        time.sleep(0.01)
```

```
            self.pos_count = 0
        def idt_motor(self):
            if self.pushIO == 20 and self.pullIO == 21:
                self.pos_motor_flag = 4
            if self.pushIO == 14 and self.pullIO == 15:
                self.pos_motor_flag = 1
            if self.pushIO == 23 and self.pullIO == 24:
                self.pos_motor_flag = 1
        def position_push(self):
            self.idt_motor()
            if self.position == 0 or self.pos_speed == 0:
                distance = 0
            interval = 0
        else:
                distance = int(1000*self.position/self.pos_motor_flag)
            interval = 0.0005*60/self.pos_speed*self.pos_motor_flag
            for i in range(0, distance):
                GPIO.output(self.pushIO, False)
                time.sleep(interval)
                GPIO.output(self.pushIO, True)
                time.sleep(interval)
        def position_pull(self):
            self.idt_motor()
            if self.position == 0 or self.pos_speed == 0:
                distance = 0
                interval = 0
            else:
                distance = int(abs(1000*self.position/self.pos_motor_flag))
                interval = 0.0005*60/self.pos_speed*self.pos_motor_flag
            for i in range(0, distance):
                GPIO.output(self.pullIO, False)
                time.sleep(interval)
                GPIO.output(self.pullIO, True)
                time.sleep(interval)
        def get_position_sleep_time(self):
            if self.position == 0 or self.pos_speed == 0:
                return 0.001
            else:
                return abs(self.position*60/self.pos_speed)
def get_position_count_sleep_time(self):
            if self.pos_count == 0 or self.pos_speed == 0:
                return 0.001
            else:
                return abs(self.pos_count*self.re_vol_pos*60/self.pos_speed)
from ctypes import *
import time
BOOL = c_int
DWORD = c_ulong
HANDLE = c_void_p
UINT = c_uint
```

```
CHAR = c_char_p
USHORT = c_ushort
LONG = c_long
INT = c_int
class MaxonMotor(object):
    BOOL = c_int
    DWORD = c_ulong
    HANDLE = c_void_p
    UINT = c_uint
    CHAR = c_char_p
    USHORT = c_ushort
    LONG = c_long
    INT = c_int
    def __init__(self, RMNodeId, pDeviceName, pProtocolStackName, pInterfaceName, pPortName, lBaudrate):
        self.RMNodeId = USHORT(RMNodeId)
        self.pDeviceName = CHAR(pDeviceName)
        self.pProtocolStackName = CHAR(pProtocolStackName)
        self.pInterfaceName = CHAR(pInterfaceName)
        self.pPortName = CHAR(pPortName)
        self.lBaudrate = UINT(lBaudrate)
        self.RMHandle = HANDLE(0)
        self.errorCode = UINT(0)
        self.lTimeout = UINT(0)
        self.rmPosition = INT(0)
        self.rmVelosity = INT(0)
        self.rotationMotor = cdll.LoadLibrary("libEposCmd.so")
        self.OpenDevice = self.rotationMotor.VCS_OpenDevice
        self.OpenDevice.argtypes = [CHAR, CHAR, CHAR, CHAR, POINTER(UINT)]
        self.OpenDevice.restype = HANDLE
        self.GetProtocolStackSettings = self.rotationMotor.VCS_GetProtocolStackSettings
        self.GetProtocolStackSettings.argtypes = [HANDLE, POINTER(UINT), POINTER(UINT), POINTER(UINT)]
        self.GetProtocolStackSettings.restype = BOOL
        self.SetProtocolStackSettings = self.rotationMotor.VCS_SetProtocolStackSettings
        self.SetProtocolStackSettings.argtypes = [HANDLE, UINT, UINT, POINTER(UINT)]
        self.SetProtocolStackSettings.restype = BOOL
        self.SetEnableState = self.rotationMotor.VCS_SetEnableState
        self.SetEnableState.argtypes = [HANDLE, USHORT, POINTER(UINT)]
        self.SetEnableState.restype = BOOL
        self.GetEnableState = self.rotationMotor.VCS_GetEnableState
        self.GetEnableState.argtypes = [HANDLE, USHORT, POINTER(BOOL), POINTER(UINT)]
        self.GetEnableState.restype = BOOL
        self.GetFaultState = self.rotationMotor.VCS_GetFaultState
        self.GetFaultState.argtypes = [HANDLE, USHORT, POINTER(BOOL), POINTER(UINT)]
        self.GetFaultState.restype = BOOL
        self.ClearFault = self.rotationMotor.VCS_ClearFault
        self.ClearFault.argtypes = [HANDLE, USHORT, POINTER(UINT)]
        self.ClearFault.restype = BOOL
        self.MoveWithVelocity = self.rotationMotor.VCS_MoveWithVelocity
        self.MoveWithVelocity.argtypes = [HANDLE, USHORT, LONG, POINTER(UINT)]
        self.MoveWithVelocity.restype = BOOL
        self.ActivateProfileVelocityMode =
self.rotationMotor.VCS_ActivateProfileVelocityMode
        self.ActivateProfileVelocityMode.argtypes = [HANDLE, USHORT, POINTER(UINT)]
        self.ActivateProfileVelocityMode.restype = BOOL
        self.HaltVelocityMovement = self.rotationMotor.VCS_HaltVelocityMovement
        self.HaltVelocityMovement.argtypes = [HANDLE, USHORT, POINTER(UINT)]
        self.HaltVelocityMovement.restype = BOOL
        self.ActivateProfilePositionMode =
self.rotationMotor.VCS_ActivateProfilePositionMode
        self.ActivateProfilePositionMode.argtypes = [HANDLE, USHORT, POINTER(UINT)]
        self.ActivateProfilePositionMode.restype = BOOL
        self.SetPositionProfile = self.rotationMotor.VCS_SetPositionProfile
        self.SetPositionProfile.argtypes = [HANDLE, USHORT, UINT, UINT, UINT,
POINTER(UINT)]
        self.SetPositionProfile.restype = BOOL
        self.MoveToPosition = self.rotationMotor.VCS_MoveToPosition
        self.MoveToPosition.argtypes = [HANDLE, USHORT, LONG, INT, INT,
POINTER(UINT)]
        self.MoveToPosition.restype = BOOL
        self.HaltPositionMovement = self.rotationMotor.VCS_HaltPositionMovement
        self.HaltPositionMovement.argtypes = [HANDLE, USHORT, POINTER(UINT)]
        self.HaltPositionMovement.restype = BOOL
                self.SetMaxProfileVelocity = self.rotationMotor.VCS_SetMaxProfileVelocity
        self.SetMaxProfileVelocity.argtypes = [HANDLE, USHORT, UINT, POINTER(UINT)]
        self.SetMaxProfileVelocity.restype = BOOL
        self.GetPosition = self.rotationMotor.VCS_GetPositionIs
        self.GetPosition.argtypes = [HANDLE, USHORT, POINTER(INT), POINTER(UINT)]
        self.GetPosition.restype = BOOL
        self.GetVelocity = self.rotationMotor.VCS_GetVelocityIs
        self.GetVelocity.argtypes = [HANDLE, USHORT, POINTER(INT), POINTER(UINT)]
        self.GetVelocity.restype = BOOL
        self.CloseDevice = self.rotationMotor.VCS_CloseDevice
        self.CloseDevice.argtypes = [HANDLE, POINTER(UINT)]
        self.CloseDevice.restype = BOOL
        self.CloseAllDevices = self.rotationMotor.VCS_CloseAllDevices
        self.CloseAllDevices.argtypes = [POINTER(UINT)]
        self.CloseAllDevices.restype = BOOL
        self.open_device()
    def open_device(self):
        Result = 0
        oIsFault = BOOL(0)
        oIsEnabled = BOOL(0)
        print "Open Device-----"
        self.RMHandle = self.OpenDevice(self.pDeviceName, self.pProtocolStackName,
self.pInterfaceName, self.pPortName, byref(self.errorCode))
              print self.RMHandle, self.errorCode.value
        if self.max_speed() == 0:
            return Result
        self.ClearFault(self.RMHandle, self.RMNodeId, byref(self.errorCode))
        self.SetEnableState(self.RMHandle, self.RMNodeId, byref(self.errorCode))
        return Result
    def max_speed(self):
        Result = 0
        if self.SetMaxProfileVelocity(self.RMHandle, self.RMNodeId, UINT(90),
byref(self.errorCode)) != BOOL(0):
            Result = 1
        return Result
    def rm_move_to_position(self, positionModeSpeed, targetRelativePosition):
        Result = 1
        positionModeAcceleration = UINT(1000)
        positionModeDeceleration = UINT(1000)
byref(self.errorCode))
        self.ActivateProfilePositionMode(self.RMHandle, self.RMNodeId,
byref(self.errorCode))
        self.SetPositionProfile(self.RMHandle, self.RMNodeId, UINT(positionModeSpeed),
positionModeAcceleration, positionModeDeceleration, byref(self.errorCode))
        self.MoveToPosition(self.RMHandle, self.RMNodeId, LONG(targetRelativePosition),
INT(0), INT(1), byref(self.errorCode))
        return Result
    def rm_halt_position_mode(self):
        Result = 1
        if self.HaltPositionMovement(self.RMHandle, self.RMNodeId, byref(self.errorCode))
== BOOL(0):
            Result = 0
    def rm_speed_and_position(self):
        Result = 0
        if self.GetPosition(self.RMHandle, self.RMNodeId, byref(self.rmPosition),
byref(self.errorCode)) != BOOL(0):
            if self.GetVelocity(self.RMHandle, self.RMNodeId, byref(self.rmVelosity),
byref(self.errorCode)) != BOOL(0):
                Result = 1
        return Result
    def close_device(self):
        Result = 0
        if self.CloseDevice(self.RMHandle, byref(self.errorCode)) != BOOL(0):
            Result = 1
    return Result
    def rm_move(self, TargetVelocity):
        Result = 1
        positionModeAcceleration = UINT(1000)
        positionModeDeceleration = UINT(1000)
        self.ActivateProfileVelocityMode(self.RMHandle, self.RMNodeId,
byref(self.errorCode))
        self.MoveWithVelocity(self.RMHandle, self.RMNodeId, LONG(TargetVelocity),
byref(self.errorCode))
    return Result
```