PXIe接口API使用说明

初始化部分

代码块

```
import ctypes,os
import chardet
import binascii
import sys
from sys import getsizeof
import numpy as np
import math
import time
import matplotlib.pyplot as plt
import scipy as sp
from scipy.optimize import curve_fit
from scipy.fftpack import fft,ifft
from scipy import signal
# import librosa
import numpy.ctypeslib as npct
from scipy.stats import norm
from scipy.stats.stats import SpearmanrResult
DRIVER_ELEC = ctypes.cdll.LoadLibrary('D:/qzz/Driver_isa.dll')

```

- 下载vs2022，需要其中的一些环境依赖
- 首先导入所需要使用的函数包
- 将C++编译生成的动态链接库导入，也就是18行的DRIVER_ELEC，在调用dll库中函数时需要DRIVER_ELEC作为前缀

代码块

```
def reverse_every_16_bytes(byte_array):
    # 将字节数组分成每 16 字节一组
    chunks = [byte_array[i:i+16] for i in range(0, len(byte_array), 16)]

    # 对每一组进行翻转
    reversed_chunks = [chunk[::-1] for chunk in chunks]

    # 将翻转后的组合并回一个字节数组
    return b''.join(reversed_chunks)
```

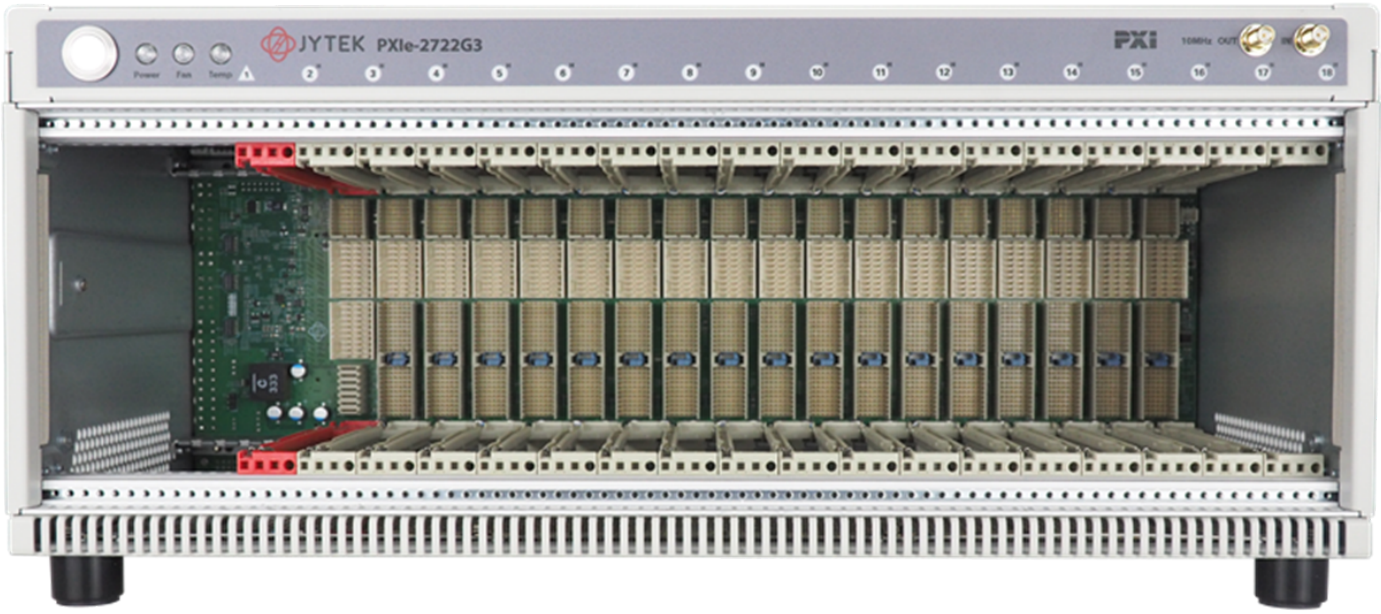
```python

def print_hex(bytes):
    l = [hex(int(i)) for i in bytes]
    print(l)

def extract_16byte_numbers(data):
    """
    将数据每 16 个字节提取为一个大数，组成数组返回。

    参数:
    - data: bytes 或 bytearray 类型，表示输入的二进制数据。

    返回:
    - 一个数组，每个元素是从 16 字节解析出的整数。
    """
    # 检查输入长度是否是 16 的倍数
    if len(data) % 16 != 0:
        raise ValueError("数据长度不是 16 的倍数")

    numbers = []
    for i in range(0, len(data), 16):
        # 提取 16 字节
        chunk = data[i:i+16]
        # 将 16 字节转化为无符号整数 (big-endian)
        number = int.from_bytes(chunk, byteorder='big', signed=False)
        numbers.append(number)

    return numbers
```

- 这里由于pcie传输的数据是二进制并且字节序为大端序（高位字节排放在内存的低地址端），因此需要先用reverse_every_16bytes进行翻转。
- 另外由于实际上是128bit为一组数据，所以要对128bit的数据进行合并，也就是使用extract_16bytes进行处理得到的才是上传的数据

代码块

```
awg0 = '0005'.encode('utf-8')
awg1 = '000d'.encode('utf-8')
ttl0 = '0007'.encode('utf-8')
```

- 对于每一块板卡的命名可以按照上述方式进行，其中‘0005’这样的序号取决于在机箱上对应插槽的编号，下图中机箱的上边缘的编号即为需要的编号。
- 进行编号之后后续的函数就都可以使用上述的方式进行各个板卡的控制



代码块

```
ttl_ch0 = 0
ttl_ch1 = 0
ttl_ch2 = 1
ttl_ch3 = 1
ttl_ch4 = 1
ttl_ch5 = 1
ttl_ch6 = 1
ttl_ch7 = 1
ttl_ch8 = 1
ttl_ch9 = 1
ttl_ch10 = 1
ttl_ch11 = 1
ttl_ch12 = 1
ttl_ch13 = 1
ttl_ch14 = 1
ttl_ch15 = 1
ttl_ch16 = 1
ttl_ch17 = 1
ttl_ch18 = 1
ttl_ch19 = 1
ttl_ch20 = 1
ttl_ch21 = 1
ttl_ch22 = 1
ttl_ch23 = 1
ttl_ch24 = 1
ttl_ch25 = 1
ttl_ch26 = 1
ttl_ch27 = 1
ttl_ch28 = 1
```

```
ttl_ch29 = 1
ttl_ch30 = 1
ttl_ch31 = 1
ttl_in_out_config =
[ttl_ch0,ttl_ch1,ttl_ch2,ttl_ch3,ttl_ch4,ttl_ch5,ttl_ch6,ttl_ch7,

ttl_ch8,ttl_ch9,ttl_ch10,ttl_ch11,ttl_ch12,ttl_ch13,ttl_ch14,ttl_ch15,

ttl_ch16,ttl_ch17,ttl_ch18,ttl_ch19,ttl_ch20,ttl_ch21,ttl_ch22,ttl_ch23,

ttl_ch24,ttl_ch25,ttl_ch26,ttl_ch27,ttl_ch28,ttl_ch29,ttl_ch30,ttl_ch31]
def ttl_config_init(bits):
    number = 0
    for i in range(32):
        number += bits[i] << i
    return number
```

- 这部分代码是为了定义ttl板的32个通道的输入输出方向，其中ttl=1是输出，ttl=0是输入
- ttl_config_init是将这32个通道的1或者0的状态写入到一个32位数字中，方便进行之后的命令下发

代码块

```
ttl_config_init_num = ttl_config_init(ttl_in_out_config)
DRIVER_ELEC.sys_ttl_status(ttl0,ttl_config_init_num) #设置ttl的32个通道的输入输出状态，例程中全部设置为输出
```

- 调用的方式如上所示，首先通过ttl_config_init_num进行计算，将计算结果放入到sys_ttl_status中进行初始化

机箱初次上电校准

代码块

```
DRIVER_ELEC.awg_sync_delay_load(awg0, 90)
DRIVER_ELEC.awg_calib_status(awg0)
DRIVER_ELEC.awg_sync_delay_again(awg0)
DRIVER_ELEC.awg_sync_delay_success(awg0) #执行本指令意思是调整延时完成，可以进行正常操作
```

- 在机箱第一次使用时需要使用以上四个函数进行机箱数据同步校准
- DRIVER_ELEC.awg_sync_delay_load(awg0, 90)，这个函数输入同步延迟值，延迟值在0到200之间选择，通常校准阶段以10为步长进行扫描

- DRIVER_ELEC.awg_calib_status(awg0)，这个函数用于在输入同步延迟值之后观察同步状态，如果同步出错，那么会返回1，如果同步成功则返回0
- DRIVER_ELEC.awg_sync_delay_again(awg0)，这个函数用于再次输入新的同步延迟值之前的复位
- DRIVER_ELEC.awg_sync_delay_success(awg0)，这个函数用于通知板卡校准完成
- 因此一般使用的扫描方式如下

代码块

```
for i in range(0,21):
    DRIVER_ELEC.awg_sync_delay_load(awg1, i*10)
    time.sleep(10)
    a = DRIVER_ELEC.awg_calib_status(awg1)
    print(i*10,a)
    DRIVER_ELEC.awg_sync_delay_again(awg1)
    time.sleep(1)
```

- 从0到200进行扫描，每次间隔10s之后查询同步状态，并且打印出相应状态
- 之后再复位输入新的同步延迟值
- 最终得到的输出类似于：

代码块

```
0 1
10 1
20 1
30 1
40 0
50 0
60 0
70 0
80 0
90 0
100 0
110 0
120 0
130 0
140 0
150 0
160 0
170 0
180 0
190 0
200 1
```

- 选择0的最中间位置作为最终的同步延迟输入，如上就应该选择115作为最终的同步延迟，最终输入就只需要如下操作：

代码块

```
DRIVER_ELEC.awg_sync_delay_load(awg0, 115) #调整通道延时
time.sleep(0.1)
DRIVER_ELEC.awg_sync_delay_success(awg0) #执行本指令意思是调整延时完成，可以进行正常操作
```

- 上述是单个板的操作，多个板只需要重复即可，在完成机箱第一次上电之后，可以将最终的同步延迟进行记录，同一个机箱下一次开机可以直接输入最终的同步延迟值而无需重复进行校准

操作流程

复位

代码块

```
#系统复位
DRIVER_ELEC.sys_reset(ttl0)
#设置ttl的32个通道的输入输出状态，例程中全部设置为输出
ttl_config_init_num = ttl_config_init(ttl_in_out_config)
DRIVER_ELEC.sys_ttl_status(ttl0,ttl_config_init_num) #设置ttl的32个通道的输入输出状态，例程中全部设置为输出
```

- 首先进行系统的复位操作，复位操作发送到ttl上，再由TTL转发到TCM板，TCM同步路由到所有的板卡，因此DRIVER_ELEC.sys_reset(ttl0)这个函数中的变量是ttl0
- 接下来就是设置TTL各通道的输入输出状态，前面已经详细描述，此处不做赘述
- 目前已经将复位功能移动到TCM板上进行，因此板号应该切换成tcm板的板号，在18槽机箱中TCM的板号一直是10也就是

代码块

```
tcm = '000a'.encode('utf-8')
DRIVER_ELEC.sys_reset(tcm)
```

加载序列

代码块

```
#各部分初始化
DRIVER_ELEC.ttl_init(ttl0, r'D:\qzz\riscq_test\TTL_32bit.txt'.encode('utf-8'))
time.sleep(0.1)
DRIVER_ELEC.ttl_init(ttl0, r'D:\qzz\riscq_test\TTL_16bit.txt'.encode('utf-8'))
time.sleep(0.1)
DRIVER_ELEC.awg_ddr_init(awg1, r'D:\qzz\riscq_test\dds_32bit.txt'.encode('utf-8'))
time.sleep(0.1)
DRIVER_ELEC.awg_ddr_init(awg1, r'D:\qzz\riscq_test\dds_param.txt'.encode('utf-8'))
time.sleep(0.1)
```

- 将序列编译器产生的文件依次下发到各个板卡上，对ttl板卡使用DRIVER_ELEC.ttl_init(ttl0, r'D:\qzz\riscq_test\TTL_32bit.txt'.encode('utf-8'))这个函数，其中第一个参数是板卡号，第二个参数是文件名称
- DRIVER_ELEC.awg_ddr_init(awg1, r'D:\qzz\riscq_test\dds_32bit.txt'.encode('utf-8'))是AWG板卡文件发送的函数，与TTL板卡文件发送函数类似
- 注意：在文件发送之间需要一定的时间间隔，否则会发送失败，目前采用的间隔是100ms

序列执行

代码块

```
#系统开始运行
DRIVER_ELEC.sys_trigger(ttl0)
```

- 同样是处于同步考虑，序列运行指令和序列复位指令一样都是在TTL板卡上接收，再由TCM板卡同步路由到所有板卡实现同步，方式如上所示：
- 同样的这部分功能也已经迁移到TCM板卡上，与reset相同

代码块

```
DRIVER_ELEC.sys_trigger(tcm)
```

note：之所以选择将reset功能和trigger功能迁移到TCM是由于之前我们只有一个输入模块，也就是TTL，但是未来会加入图像板进行进一步的数据收集，因此，TTL板不再适合作为主控，主控应该由TCM承担，并且TCM和各板卡相连，可以接收多个板卡的反馈，并且进行仲裁之后送入到其余板卡中。

数据收集

```
代码块
buffer = ctypes.create_string_buffer(4096)
DRIVER_ELEC.start_streaming(ttl0, 200000)
max_iterations = 100
iteration = 0
try:
    while iteration < max_iterations:
        n = DRIVER_ELEC.get_latest_data(buffer, 4096)
        if n > 0:
            data = np.frombuffer(buffer.raw[:n], dtype = np.uint8)
            print(data)
        else:
            time.sleep(1)
            print("no data")
        iteration += 1
finally:
    DRIVER_ELEC.stop_streaming()
```

- 系统需要对TTL的计数进行实时的显示（绘图），因此将采数过程作为一个后台线程，如上所示：DRIVER_ELEC.start_streaming(ttl0, 200000)这个函数就是开启采数线程的命令，其中第一个参数是板卡号，第二个参数是超时时间，如果达到这个时间还没有结束进程，那么采数进程会自动结束，其单位是ms。
- DRIVER_ELEC.get_latest_data(buffer, 4096)，这个函数用来实时取出采数进程取出的数据，可以先用之前描述的reverse_every_16_bytes和extract_16byte_numbers两个函数进行处理，处理后得到的即为计数值，可以再python中进行数据的实时处理，并且刷新的时间在python中决定，如上述代码中如果是空的就等1s再读取。
- DRIVER_ELEC.stop_streaming()这个函数用于结束采数的后台线程

系统状态查询

DAC上电同步状态查询

```
代码块
dac_status = DRIVER_ELEC.awg_dac_sync_success(awg0)
if dac_status == 0x30303030:
    print('dac sync success')
else:
    print('dac sync failed')
```

- 对于单个DAC来说，上电同步状态完成，从板子上查询到的状态应该是0x30，由于一次查询能查到32bit数据，因此，将四片DAC的状态合成一个32bit数放在同一个寄存器中，因此，查询到的数字是0x30303030时表示DAC上电同步状态正常

DAC当前输出频率

代码块

```
freq = DRIVER_ELEC.awg_dac_freq(awg0, 1)
freq = freq/2**30*250
print(freq)
```

- DRIVER_ELEC.awg_dac_freq(awg0, 1)，参数中包含两个变量，第一个是板号，第二个是DAC通道号，可以是0-23，其中0-5是第一个DAC输出的六个频率，以此类推
- 由于设定的频率字的位宽是32位，但是会出现频率大于250M的情况，因此实际位宽是34位，而查询寄存器的位宽是32位，因此截取频率字的高32位进行读出，由于去掉了低两位，因此还原数据应该是freq/2^30*250

TCM设置接收反馈的通道

目前TCM可以配置两个通道来接收板卡（TTL和CCG（图像板卡）各一个）的反馈，并且将反馈信息广播到其余所有板卡，但是很多情况下只需要顺序执行，或者只需要一个通道（比如只使用了TTL板卡进行反馈的情况），在这些情况下，就需要对两个反馈通道进行灵活的配置，避免产生错误：

代码块

```
DRIVER_ELEC.decoder_enable(tcm, 1, 1) #这是两个通道都打开的情况
DRIVER_ELEC.decoder_enable(tcm, 1, 0) #这是开启通道0的反馈，关闭通道1反馈的情况
```

Driver_ttl bool decoder_enable(char* subid, uint32_t channel_0_en, uint32_t channel_1_en)，函数原型是这样的，其中后两个参数分别是两个通道各自的使用，默认情况下都关闭，对应参数设置为1的时候通道打开，也就是接收该通道的反馈信息，并且进行广播