# 接口说明

## 1. 连接

连接模块的`Connection`类用于建立应用与设备之间的连接。

### Connection

```
class Connection():
    def __init__(self, sensorName, applicationName):
        ...
```

其构造函数需要`sensorName`和`applicationName`两个字符串，其中`sensorName`是设备的IP地址，例如"192.168.1.10"，`applicationName`是应用的名字，可自定义一个字符串。

`Connection`支持python `with`操作，可在`with`块退出时自动释放连接。

> 在开发时，可以在一开始就建立一个Connection的实例，在程序运行期间一直保留该实例，以便为各种数据采集任务提供连接句柄，直到程序退出时，再清除该实例。

## 2. 频谱采集

频谱模块包括`SpectrumSnapshot`和`SpectrumScaner`两个控制频谱采集的类。其中`SpectrumSnapshot`根据输入的扫描参数只扫描一次，相当于给特定的频段拍了一张快照。与之相比，`SpectrumScaner`会对指定的频段连续扫描，直到向它发送扫描停止的指令。

### snapshot、scaner通用属性

`startFrequency` 只读，起始频率

`stopFrequency` 只读，终止频率

`totalPoints` 只读，一次扫描的频点个数

`totalSegments` 只读，一次扫描所扫过的频段个数

### SpectrumSnapshot

```
gene(self, start = 88e6, stop = 108e6, rbw = 10e3, **kwargs):
```

根据start, stop等参数生成扫描参数，启动接收机采集，生成频谱结果。

- `start` 起始频率，单位Hz
- `stop` 终止频率，单位Hz
- `rbw` 分辨率带宽，单位Hz

- `kwargs`，其它参数项：
  - `avg` 平均次数 [0, 64]，默认0
  - `gain` 前置放大器开关 True or False，默认False
  - `att` 衰减器[0,30]，默认0
- 返回值：`(header, spectrum)`，其中header为数据头，是`SegmentData`类型的实例，包含数据产生的时间、方位等信息。spectrum为float[]，为数据体。

## SpectrumScaner

```
config(self, start, stop, rbw, avg = 0, gainSwitch = False, attenuation = 0)
```

配置扫描，参数含义同`SpectrumSnapshot.gene()`

```
start(self, sweepCount = 0, sweepInterval = 100)
```

启动扫描

- `sweepCount` 扫描次数，默认为0，即无限次
- `sweepInterval` 扫描间隔，相邻两次扫描之间的时间间隔，单位毫秒，默认100毫秒

```
abort(self)
```

终止扫描

```
reset(self)
```

重置扫描，释放测量句柄

`SpectrumScaner`支持python `with`，在`with`块退出时自动释放数据连接。

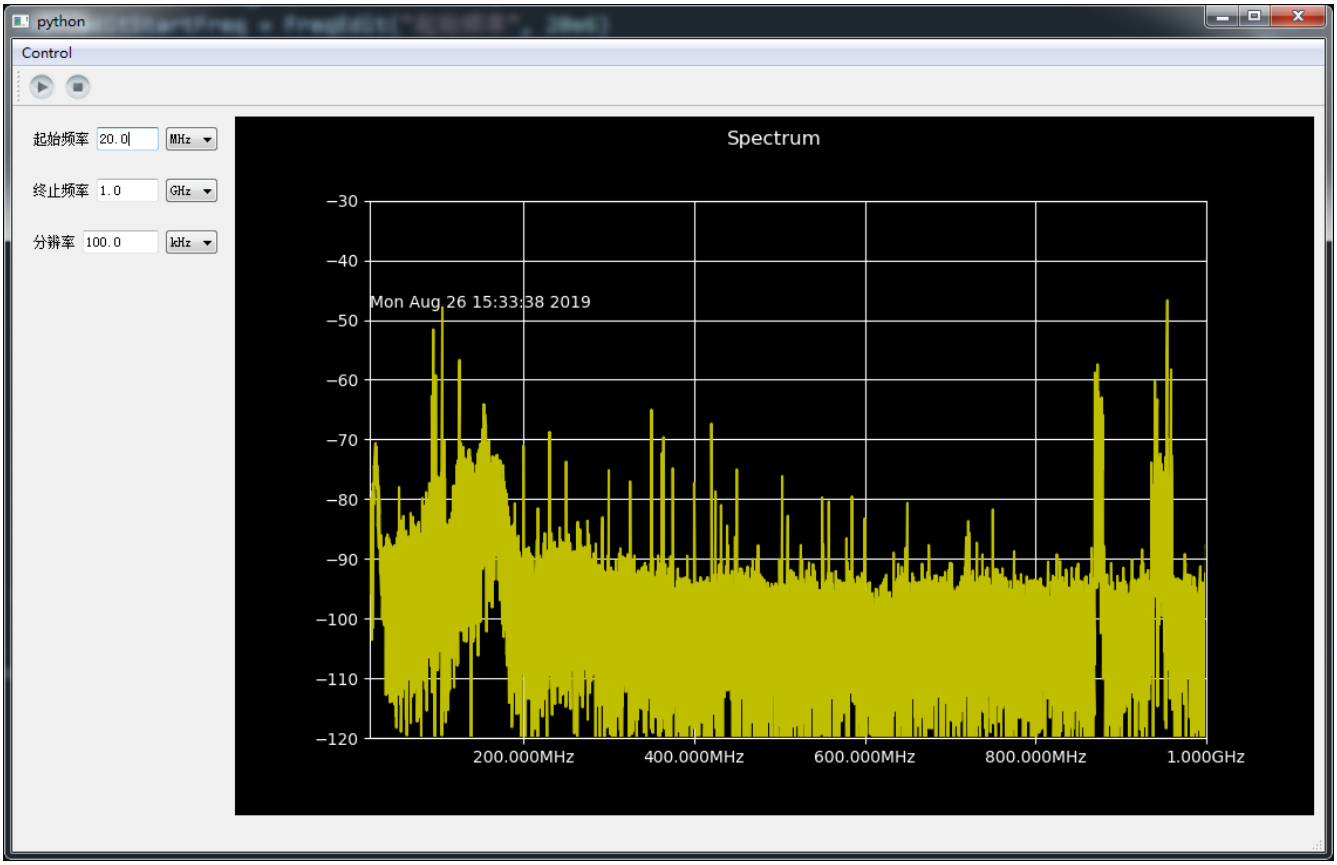
并且支持迭代操作，在调用start()成功后，可以使用for、next等方法循环获取结果。

一个典型的扫描过程如下：

```
with SpectrumScaner(sensorHandle) as scaner:
    scaner.config(88e6, 108e6, 100e3)   #配置扫描频率范围88MHz-108MHz，分辨率100kHz
    if not scaner.start():
        return
    startTime = time.time()
    for header, spectrum in scaner:
        doSomeThing(header, spectrum)   #此处对获取的结果进行处理
        if timer.time() - startTime > 100: #如果扫描了100秒，停止扫描
            scaner.abort()              #会异步触发StopIteration异常，从而退出循环
```

具体示例可见`example/spectrum_sweep.py`，示例依赖`matplotlib`和`pyqt5`包，请确定已安装，再运行。

在窗口左侧设置好起始终止频率，分辨率，然后点击三角图标运行。




# 3. IQ采集

IQ采集模块包括IQ采集类 `IQSnapshot`。 `IQSnapshot`非常类似于频谱采集模块的 `SpectrumSnapshot`，都是根据传入的扫描参数启动采集，采集完成后返回采集结果。

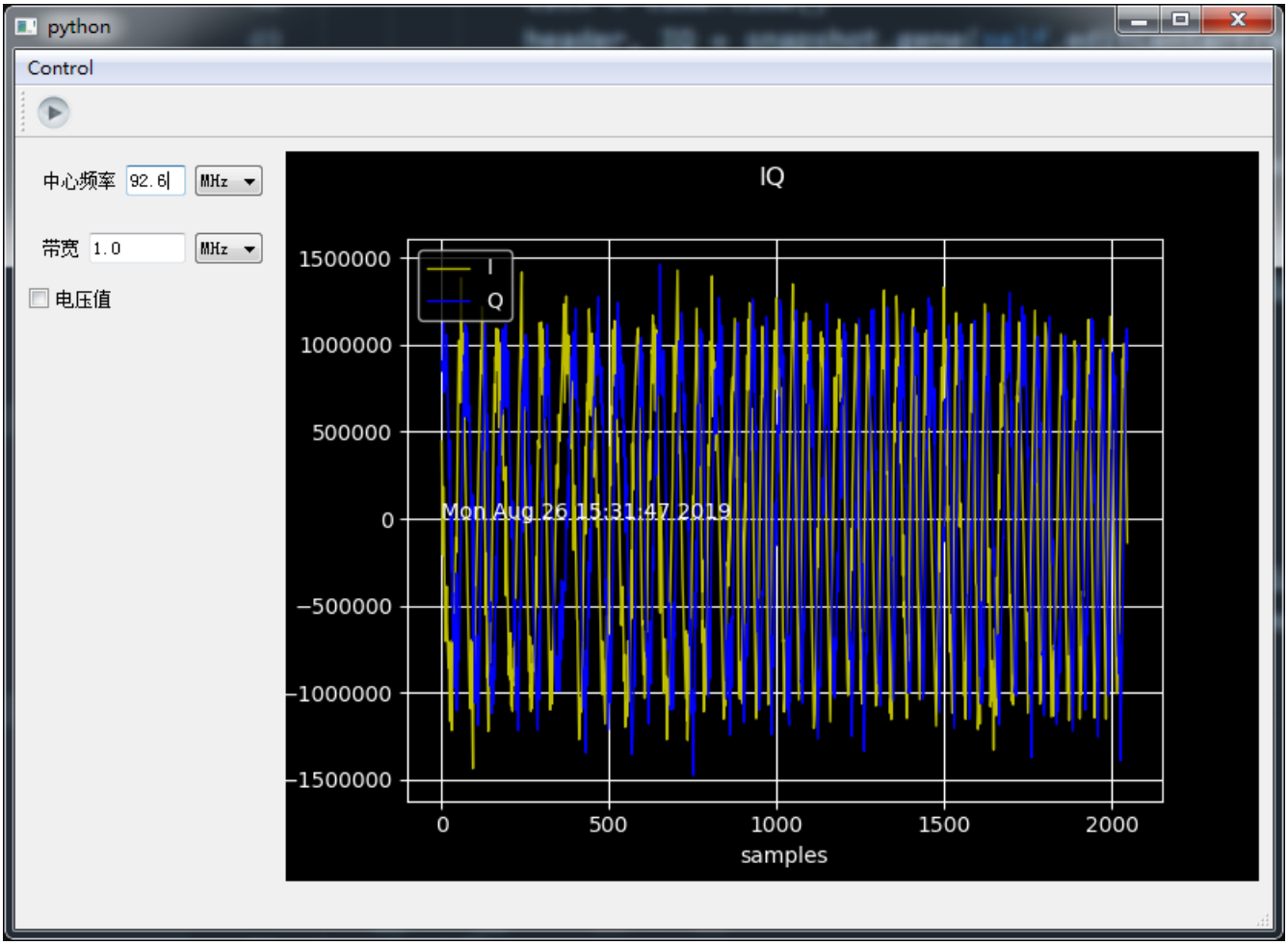
## IQSnapshot

```
def gene(self, center, span, **kwargs)
```

根据center, span等参数生成扫描参数，启动接收机采集，生成结果。

- `center`中心频率，单位Hz
- `span`分析带宽，单位Hz，3900A可取[10kHz, 20MHz]，3900D/E/F可取[10kHz, 40MHz]
- `kwargs`，其它参数项：
  - `numSweeps`扫描次数 [0, N]，最大可取uint64的max值，0为无限次采集，默认1
  - `numTransferSamples`每次采集的采样点数，可取256, 512, 1024, 2048等2的整数次幂，最大可取2048，默认2048
  - `numBlocks` 采集的频点数，默认为1
  - `gain`前置放大器开关 True or False，默认False
  - `att` 衰减器[0,30]，默认0
- 返回值：`(header, IQ)`，其中header为数据头，是`IQSegmentData`类型的实例，包含数据产生的中心频率、采样率、时间、方位等信息。IQ为int[]，为数据体，长度为`numTransferSamples`的两倍，这是因为每个采样点包含两个数据，分别为I和Q，数据在列表中交叉排列，双数索引上为I值，单数索引上为Q值。

具体示例可见`example/IQ_snapshot.py`




# 4. FM/AM解调

FM/AM解调模块包括解调器`AnalogDemodulator`和解调频谱控制类`AnalogDemodulatorSpectrumScaner`。其中`AnalogDemodulator`控制接收机输出音频流，`AnalogDemodulatorSpectrumScaner`控制接收机并行输出分析带宽内的频谱。

## AnalogDemodulator

### 属性

`audioSampleRate` 只读，音频采样率

### 接口

```
def config(self, center, analyseBw, gainSwitch = False, attenuation = 0)
```

配置解调器参数

- `center` 中心频率，单位Hz
- `analyseBw` 分析带宽，单位Hz，3900A可取[10kHz, 20MHz]，3900D/E/F可取[10kHz, 40MHz]

- `gainSwitch` 前放开关
- attenuation 衰减设置[0,30]

```
start(self)
```

启动解调，返回True成功，False失败

```
changeTuner(self, center, bw)
```

切换解调频率和解调带宽，仅可在start()之后调用

- `center` 待切换的解调频点，应在config()调用参数的 `center ± analyseBw /2` 范围之内
- `bw` 新的解调带宽，典型值100kHz-200kHz
- 返回True成功，False失败

`AnalogDemodulator` 支持python `with`，在 `with` 块退出时自动释放数据连接。

并且支持迭代操作，在调用start()成功后，可以使用for、next等方法循环获取结果。

## AnalogDemodulatorSpectrumScaner

在成功启动解调器之后，可以创建一个 `AnalogDemodulatorSpectrumScaner` 对象，并行采集分析带宽内的频谱。
当然如果不关注频谱，只解调的话也是可以的，可跳过下面的说明。

### 属性

`center` 只读， 中心频率

`span` 只读，分析带宽

`points` 只读，频谱结果的有效点数

### 接口

```
def __init__(self, demodulator)
```

该类构造函数需要传入解调器对象

```
def start(self)
```

启动频谱扫描，仅可在demodulator.start()之后调用

```
def abort(self)
```

终止频谱扫描

`AnalogDemodulatorSpectrumScaner` 支持python `with`，在 `with` 块退出时自动释放数据连接。

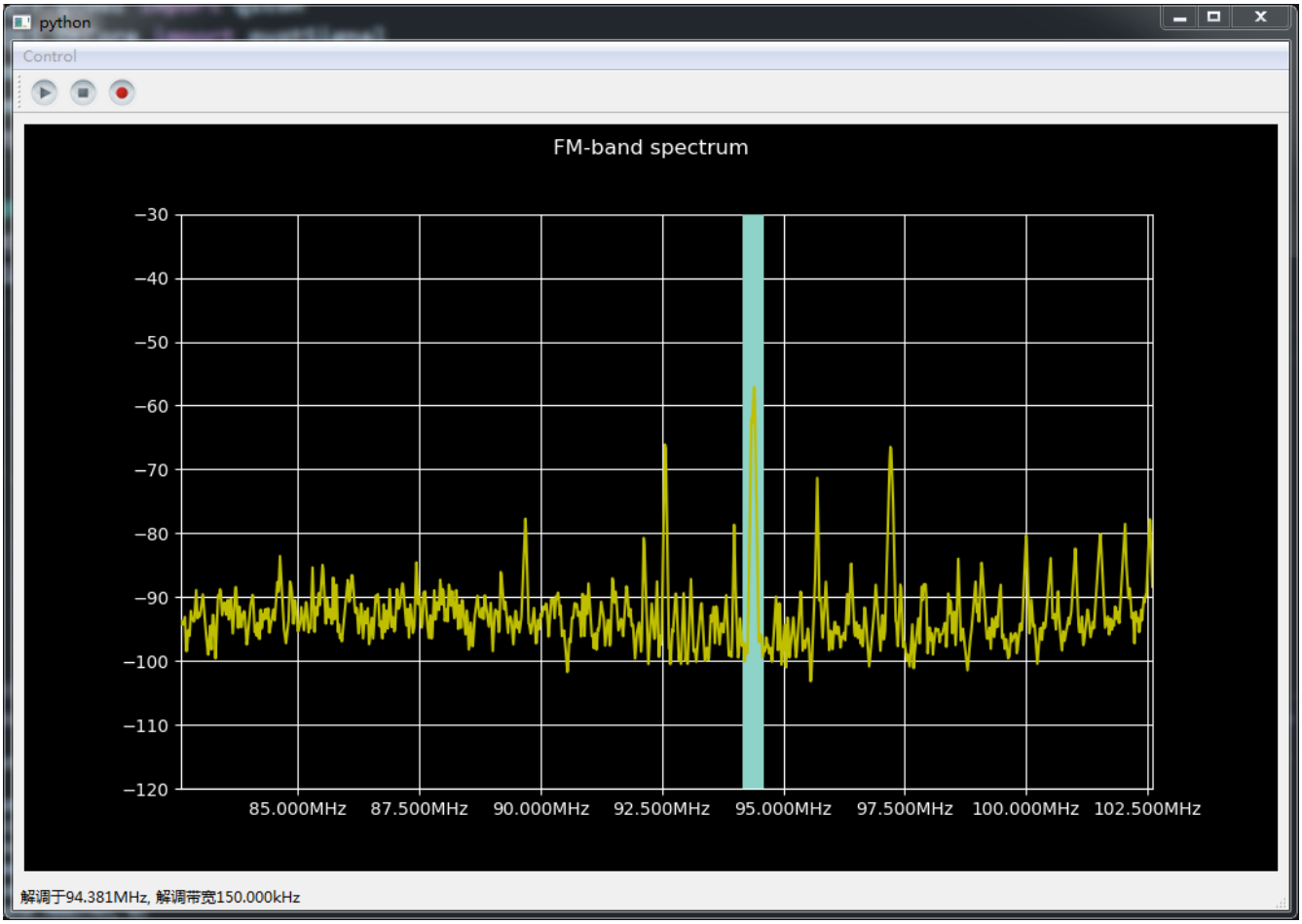
并且支持迭代操作，在调用start()成功后，可以使用for、next等方法循环获取结果。

一个典型的解调加频谱扫描的过程如下：

```
import sys
import eisal
from time import time
from time import sleep
import threading

def test():
    with eisal.Connection("192.168.1.88", "demod") as connection:
        with eisal.AnalogDemodulator(connection.native_handle()) as demodulator:
            demodulator.config(center = 92.6e6, analyseBw=20e6, gainSwitch = False,
attenuation = 0)
            if not demodulator.start():
                return
            scaner = eisal.AnalogDemodulatorSpectrumScaner(demodulator) #创建频谱扫描对象
            scaner.start()

            #定义两个线程函数，分别处理音频和频谱数据
            def audioPlayFunc():
                for header, pcmBlock in demodulator:
                    print(header.sequenceNumber)

            def spectrumAcquireFunc():
                for header, spectrum in scaner:
                    print(header.sequenceNumber)

            audioPlayThread = threading.Thread(target=audioPlayFunc)
            audioPlayThread.start()

            spectrumAcquireThread = threading.Thread(target=spectrumAcquireFunc)
            spectrumAcquireThread.start()

            sleep(10) #运行十秒后退出
            scaner.abort()
            spectrumAcquireThread.join()
            demodulator.abort()
            audioPlayThread.join()

test()
```

具体示例可见`example/analog_demod.py`，示例依赖于`pyAudioPlayer`（随`pyeisal`提供），请事先安装。

示例中，按`三角形`图标启动解调，此时声音开始播放，同时界面显示频谱，用鼠标可以拖动蓝色条带到任意信号处以解调相应的电台。按`方框`图标停止解调。按`红点`图标将当前音频保存为`.wav`格式的音频文件，文件保存在示例所在的目录下。


python
Control
FM-band spectrum
−30
−40
−50
−60
−70
−80
−90
−100
−110
−120
85.000MHz
87.500MHz
90.000MHz
92.500MHz
95.000MHz
97.500MHz
100.000MHz
102.500MHz
解调于94.381MHz, 解调带宽150.000kHz