PXView Python 解码器开发指南

基于 sigrok PD API 的自定义解码器开发 · PXLogic 官方文档 · fpgausb.com

PXView解码协议入门教程

版本	修改日期	修改人
1.0.0	2024.9.1	章鱼哥

前言

在PXView安装目录下，有一个decoders文件夹，里面有许多目录是以各种协议名称命名的。每个目录下有至少2个扩展名为.py的文件，这些都是python代码文件。在linux系统下，decoders目录位于"/usr/local/share/libsigrokdecode4DSL"下。

PXView通过底层python解释器执行python代码，对逻辑分析仪的数据进行解析，按各种算法得出需要的结果。

每个协议目录下必须存在两个文件：

1. __init__.py，用于发现模块，这个文件名左右两边各有两个下划线，这个细节要注意；

2. pd.py，用于编写主要的逻辑代码；

这两个文件如何编写，请仔细阅读下面的内容。在1.2.0以上的新版本PXView中，decoders下的有一个名为example目录为示例代码。

python入门

python语言是一个解释执行的语言。在官方网站下载并安装好python，就可以进行开发了。

新建一个文本文件，里边输入一行文字：

print('Hello,world!')

保存为 test.py，然后在命令行里输入 python test.py，将会打印“Hello,world!”

这里的python入门只是为了帮助一些读者能够顺利阅读部分协议代码，它所讲的python知识还不够全面和深入，需要读者自行通过其它方式获得python资料，以便提升自己的python编程能力。

变量定义

age = 1

name = “Tom”

数值

1, 1.11, 1000等都属数值型数据

字符串

以单引号或双引号括起来的一串字符，表示字符串，如

’ abc’

“name”

列表

[]

[1,2,3]

[1,”abc”,”name”,[7,8,9]]

列表里的元素用逗号隔开，上面的第一个列表是空列表，第二个列表全是数字，第三个列表有多种类型的元素，有数值、字符串、列表。

a = [1,2,3]

变量a是一个列表，通过a[n]方式读取列表里的元素。n的取值从 0 开始到不超过且不等于列表长度的整数

a[2] = 666

将第 3 个元素设置成666，列表里的内容可修改

i = a[1]

取列表a的第二个元素赋给变量i

字典

d = {‘age’:20, ‘name’:’Tome’, ‘data’:[7,8,9]}

字典里的每一项用一对键和值表示。如’age’:20

d[‘name’] = ‘Same’

将字典d的name值设置成’Same’

s = d[‘name’] #取字典d的name值赋给变量s

字典里的键名可以是数字、字符串、元组等类型，如：

{1:'张三'}

{'name':''张三"}

{(1,2,3): "张三"}

元组

()

(1,2,3)

(2,’abc’)

元组跟列表一样，不同的是用()括起来，元组只能读，不能修改。

注意：当元组只有一个项时，要多打一个逗号，如：(1,)

函数

def call(): #普通函数

def call(self): #类成员函数，第一个参数是必须的

def call(a,b,c): #带三个参数的函数

新建协议

新建协议目录

找到存放所有协议的decoders目录。widnows下，它在PXView的安装目录里；在linux下，它在“/usr/local/share/libsigrokdecode4DSL”里。打开decoders目录，新建一个子目录，

并给目录取名字，要求是能体现协议名称的名字。这里，我们的示列协议名为lala。

__init__.py文件

在bala目录下新建文件“__init__.py”，加入一行如下代码并保存:

from .pd import Decoder

pd.py文件

在bala目录下，建新pd.py文件，用来编写主要的代码。

框架代码模板

以下是解码协议代码框架，写在pd.py文件里。所有协议的代码核心部分是一样的。

```
# 导出核心模块类，c代码实现的类
import sigrokdecode as srd
# 协议模块类
class Decoder(srd.Decoder):
# 这里定义类的一些全局变量，有一些是底层框架要求必须要写的，其它提根据需要 #
自己加，注意缩进，不清楚请查看python手册
# 说明需要安装的python版本
api_version = 3
# 协议标识，必须唯一，这里我们用bala，正好跟上面的bala目录一致
id = 'bala'
# 协议名称, 不一定要求跟标识一致
name = 'bala'
# 协议长名称
longname = 'bala protocal'
# 简介内容
desc = 'This is an example of protocol development'
# 开源协议
license = 'gplv2+'
# 接收的输入的数据源名，如果是多层协议一起工作，可使用上一个协议的输出名
inputs = ['logic']
# 输出的数据源名，多层协议模式下，可作为下层协议的输入数据源名
```

```
outputs = ['bala']
# 适用范围标签
tags = ['Embedded/industrial']
# 必须要绑定的通道定义，将在界面上可见，设置界面见图 1
# id:通道标识, 任意命名
# type:类型，根据需要设置一个值, -1:COMMON,0:SCLK,1:SDATA,2:ADATA
# name:标签名
# desc:该通道的说明
# 注意元组的最后个逗号不能少
channels = (
{'id': 'c1', type:0, 'name': 'c1', 'desc': 'chan1-input'},
)
# 可选通道，其它跟上面的一样
optional_channels = (
{'id': 'c2', type:0, 'name': 'c2', 'desc': 'chan 2 -input'},
)
# 提供给用户通过界面设置的参数，根据业务需要来定义
# 通过self.options[id]取值，id就是各个项的id值，比如下面的"wordsize"
options = (
# 这种参数是一个下拉列表
{'id': 'debug_bits', 'desc': 'Print each bit', 'default': 'no',
'values': ('yes', 'no')},
# 这是一个输入框
{'id': 'wordsize', 'desc': 'Data wordsize (# bus cycles)', 'default': 0},
)
# 解析结果项定义
# annotations里的每一项可以有2到3个属性，当有 3 个属性时，第一个表示类型
# 类型对应0-16个颜色，当类型范围在200-299时，将绘制边沿箭头
annotations = (
('1', 'data1', 'test data1'),
('2', 'data2', 'test data2'),
('222', 'data3', 'test data3'),
)
```

```
# 解析结果行定义
annotation_rows = (
# (0,)表示可输出第1个定义的annotations类型
('lab1', 'row1', (0,)),
# (1,2)表示可输出第1个和第2个定义的annotations类型
('lab2', 'row2', (1,2)),
)
# 构造函数，自动被调用
def __init__(self):
# 这里调用一个类成员函数，完成一些参数的初始化
self.reset()
#重置函数，在这里做一些重置和定义类私有变量工作
def reset(self):
#定义一个私有变量count
self.count = 0
# 开始执行解码任务时，由c底层代码自动调用一次
# 这里，完成一些解码结果项annotation类型的注册
# 有: OUTPUT_ANN，OUTPUT_PYTHON，OUTPUT_BINARY，OUTPUT_META
# self.register函数是c底层类提供的
def start(self):
self.out_ann = self.register(srd.OUTPUT_ANN)
# 定义一个输出函数
# a,b为采样位置的起点和终点
# ann为annotations定义的项序号
# data是一个列表，列表里有 1 到 3 个字符串，它们将显示到屏幕
# annotation输出到哪一行由annotation_rows决定
# self.out_ann就是上面注册的消息类型了
# self.put是c底层类提供的函数
def put_ann(self, a, b, ann, data):
self.put(a, b, self.out_ann, [ann, data])
# 解码函数，解码任务开始时由c底层代码调用
# 这里不断循环等待所有采样数据被处理完成
def decode(self):
```

```
while True:
(a,b) = self.wait({0:'r'})
# self.wait()可带参数，也可以不带参数，不带参数时将返回每个采样数据
# 参数{0:'r'}， 0表示匹配channels第 1 项绑定的通道，'r'表示查找向上边沿
# wait函数可传多个条件，与条件:{0:'f', 1 :'r'}, 或条件：[{0:'f'},{ 1 :'r'}]
# h:高电平，l:低电平，r:向上边沿，f:向下边沿，e:向上沿或向下沿, n:要么 0 ，要么 1
# wait函数前的变量(a,b)，数量由定义的channels里的通道数决定，包括可选通道
# optional_channels 。例如：channels和optional_channels共定义了 4 个通道，
# 则变成(a,b,c,d) = self.wait()，共四个变量
# 底层模块提供的属性：
# 1. self.samplenum 当前wait()调用匹配结束的采样点位置
# 2. self.matched 本次调用wait()后条件参数匹配结果信息，是一个uint64类型数值，
# 表示本次wait调用参数的匹配信息，通过位运算来获取具体信息。
#比如[{0:'r'},{1:'f'}]，这是个或条件，判断matched的0位和1位就知道是哪个条件匹配
#成功，如：self.matched & 0b10 == 0b10,检测的是第2位bit，也就是第二个条件{1:'f'}
```

应用示例

在上面代码框架的基础上，我们接下来实现一个简单的例子。具体是，通过解码某一通道的数据，从一个向上边沿开始到向下边沿结束，输出采样点差值信息。奇数次输出放在第二行，偶数次输出放在第一行。具体编码和说明如下，最终效果见图 3 ：

```
def decode(self):
# 次数计数
times = 0
# 向上边沿样品位置
rising_sample = 0
# 条件参数列表
flag_arr = [{0:'r'}, {0:'f'}]
# wait参数的条件序号
flag_dex = 0
# 不断循环，处理完所有数据
while True:
# 边沿条件
edge = flag_arr[flag_dex]
# 取一个条件，调用wait函数，找到符合条件的数据
```

```
# (a,b)是一个元组，里边的变量数一定要跟channel数一致
(a,b) = self.wait(edge)
# 条件在向上边沿和向下边沿中切换
# 初始是向上边沿，取到样品位置后，切换成下向边沿
if flag_dex == 0:
flag_dex = 1
# 保存向上边沿采样位置
rising_sample = self.samplenum
else:
flag_dex = 0
# 打印次数计数加 1
times += 1
# 向下边沿采样位置
falling_sample = self.samplenum
# 向下边沿和向上边沿采样位置差
v = falling_sample - rising_sample
# 转换成16进制的字符串
s = '%02X' % v
# 对times求余得值在0和1中变换,对应annotation的序号
ann = times % 2
self.put_ann(rising_sample, falling_sample, ann, [s])
```

解码模块工作原理

通过c代码和python代码的互操作，将采样数据交给python分析。经过一系列的处理，最终生成解码结果，用于显示以及供给上层协议作为分析的数据来源。解码模块的核心主要由以下部分组成：

c底层包装类Decoder

在c代码里，给python提供一个经过包装的基类，python可调用基类的一些方法，
实现调用c代码的目的。python端通过以下语句导出c代码包装的Decoder类：

```
import sigrokdecode as srd
```

python可访问的Decoder基类的方法有：

register方法

用于注册python输出到c底层的消息类型，有：

(1) OUTPUT_ANN，数据输出到屏幕

(2) OUTPUT_PYTHON，数据输出到上层协议

(3) OUTPUT_BINARY

(4) OUTPUT_META

python调用方式：

self.out_ann = self.register(srd.OUTPUT_ANN)

self.out_py = self.register(srd.OUTPUT_PYTHON)

b. put方法

输出数据到屏幕或上层协议，python调用方式： self.put(a,b,self.out_ann,[0,['abc']])

其中，a、b为采样点区间值，self.out_ann为注册的消息类型，[0,['abc']]，
0为消息类型序号，参考之前的内容；['abc']为消息内容了。

c. wait方法

获得上一次分析位置后的采样数据，可通过参数指定边沿查找条件。

调用方式： self.wait()

可指定参数，如：{0,'r'}表示第 1 个绑定的通道满足向上边沿的数据；{1,'f'}表示第 2 个绑定通道足向下边沿的数据。其它条件标志还有：h、l、e、n。

可通过多个条件组成并和或的条件。并条件如：{0:'f',1:'r'}，或条件如： [{0:'f'},{1:'r'}]

d. has_channel方法

用来叛断某个通道是否绑定，python调用方式：

self.has_channel(0)， 0 是通道序号。

e. 属性

c底层类给python提供了两个属性：

self.samplenum，wait()调用后的采样数据位置；

self.matched，wait()调用后条件参数匹配信息；

python层包装类Decoder

继承至c底层包装的类，由c底层实例化并调用python类的方法，代码如下：

class Decoder(srd.Decoder):

子类Decoder的方法有：

reset方法

这里做一些变量值的重置，以及类的私有变量的定义。变量的定义如：

self.name = 'abc'

start方法

在解码任务开始执行前，c底层代码会调用一次start函数，这里主要是做一些初始化工作，比如注册消息类型，如：

self.out_ann = self.register(srd.OUTPUT_ANN)

self.out_py = self.register(srd.OUTPUT_PYTHON)

decode方法

由c底层调用，在解码任务开始时，c底层启动一个线程，然后在线程里调用decode方法。

在这个函数里，一直循环调用wait函数，不断从c底层读取符合边沿条件的数据，如：

While True:

(a,b) = self.wait()

(a,b)元组里的变量数跟声明的chnnaels里声明的通道数一致，包括可选的通道上。

解码任务执行流程：

解码任务开始启动；

上层将采样数据分批推送到底层；

底层检测并启动一个线程，该线程调用python层的decode函数，不断处理上层推送的数据；

python层经过一系列的计算处理，生成解码结果，通过put函数输出到c底层；

当所有数据推送完成并经过python层处理，解码任务结束；

框架升级更新记录

在PXView版本1.2.0以上，更新以下功能：

end方法

python层的方法，当所有数据处理完成后会被c底层触发

self.last_samplenum属性

c底层提供的属性，其值为所有数据推送完成后最后的数据样位置。当存在end方法时，该属性将被设置

数据多种显示格式

显示的annotation数据部分，可以在2进制、16进制、8进制、10进制、ascii格式间转换。

self.put(s, e, self.out_ann, [1,['%02X' % value])

put函数将数据输出c底层，并在屏幕上显示。其中，value为要显示的数据。在输出到时需要转换为16进制的字符串。当需要让数据支持在多种格式间转换时，代码修改如下：

self.put(s, e, self.out_ann, [1,['@' + '%02X' % value])

它是通过在数据部分前加@符号，告诉c底层这一部分内容是数据部分，如： '@66FB'。

如果存在前缀文字，需要将格式部分和数据部分开，如：

['Data:{$}','D:{$}', '@66FB']，{$}是占位符，系统将数据部分格式化后替换掉占位符，就会变成：Data:66FB

图 1 （channels里定义的通道介面）

图 2 （options里定义的设置参数界面）

图 3 （解码输出内容）