## DSView解码协议教程入门

|  |  |  |
| --- | --- | --- |
| 版本 | 修改日期 | 修改人 |
| 1.0 | 2022.3.10 | 邰忠政 |
|  |  |  |

#### 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!”

1. 变量定义

age = 1

name = “Tom”

1. 数值

1, 1.11, 1000等都属数值型数据

1. 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”

1. 列表

[]

[1,2,3]

[1,”abc”,”name”,[7,8,9]]

列表里的元素用逗号隔开，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开始到不超过列表长度的整数。

列表里的元素用逗号隔开。

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

1. 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

1. 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：元组只有一个元素时，要多打一个逗号，如：(1,)

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

1. 新建协议目录

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

并给目录取名字，最好是一个能体现协议名称的名字。这里我们用bala命名，来完成以下的教程。

1. \_\_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

1. pd.py文件

　在同一个目录下，建新pd.py文件，用来编写主要的代码。

1. **框架代码模板**

所有协议的pd.py文件的代码框架都是一样的。

# 导出核心模块类，c代码实现的类

import sigrokdecode as srd

# 本协议模块类

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': 'chan２-input'},

)

# 提供给用户通过界面设置的参数，根据业务需要来定义，界面见图２

options = (

{'id': 'bit-len', 'desc': 'match bit length', 'default': 16, 'values': (8,16)},

{'id': 'mode', 'desc': 'work mode', 'default': 'up','values': ('up','low')},

)

#　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 = (

('100', 'test-data1', 'example test data1'),

('201', 'test-data2', 'example test data2'),

('1', 'test-data3', 'example test data3'),

)

# 解析结果的行定义，见图３

annotation\_rows = (

('lab1', 'the lab1', (0,)), #(0,)表示可输出第1个定义的annotations类型

('lab2', 'the lab2', (1,2)), #(1,2)表示可输出第1个和第2个定义的annotations类型

)

# 构造函数，自动被调用

def \_\_init\_\_(self):

self.reset()　# 这里调用一个类成员函数，完成一些参数的初始化

# 重置函数，在这里做一些重置和定义私有变量工作，需要哪些变量，由具体编程需 #要来定

def reset(self):

self.count = 0 #定义一个私有变量count

# 开始解码数据时，自动调用一次，由c底层代码调用

# 这里，完成注册一些解析结果项类型annotation类型

# 有: OUTPUT\_ANN，OUTPUT\_PYTHON，OUTPUT\_BINARY，OUTPUT\_META

def start(self):

self.out\_ann = self.register(srd.OUTPUT\_ANN)

# 输出函数

def put\_row1(self, a, b, value):

self.put(a, b, self.out\_ann, [0, ['{$}', '@%02X' % value]])

#第四个参数列表的０表示对应哪个annotation，至于输出在哪一行就看annotation

#的序号放在annotation\_rows中哪一行上

# a，b为采样起点和终点, value为要输出的数值

#'@%02X', 前边加@是告诉底层模块这是一个数值数据，显示格式可转化为#hex/oct/bin/acii/dec

#{$}是占位符,内容由格式化后的数值部分填充

# @特殊符号和{$}占位符的特性，只有DSView版本在1.2.0以上才支持

# []描述输出内容，第一个元素表示annotation序号,

#annotation输出在哪一行由annotation\_row决定，put函数的第四个参数是一个列表，

#列表的第一个元素表示是一个数字，表示对应哪一个annotation。列表的第二个元素又

#是一个列表，全是字符串，有１到３个，视图会根据空间决定显示哪一个。

#自定义的第二行输出函数

def put\_row2(self, a, b, value):

self.put(a, b, self.out\_ann, [1, ['{$}', '@%02X' % value]])

#第四个参数列表的１表示对应哪一个annotation

# 解码函数，触发解码逻辑，由c底层代码调用

def decode(self):

#这里不断循环等待所有采样数据被处理完成

while True:

(a,b) = self.wait({0:'r'})

# wait({0:'r'}), 0表示匹配channels定义的第１个通道，'r'表示查找向上边沿

# wait()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()前的变量(a,b)，变量数由定义的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

#则变成(a,b,c,d) = self.wait()，四个变量

#wait()不带参数时，表示匹配所有采样点数据

# 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()后所有通道的匹配结果信息，是一个uint64类型数值，

#表示０到63个通道的匹配信息，通过位运算来获取具体信息。

1. **应用示例**

在上边代码框架的基础上，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，以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):

times = 0　#打印次数

lst\_dex = self.samplenum #取初始化采样点

flag\_arr = [{0:'r'}, {0:'f'}] #匹配条件数组

flag\_dex = 0 #当前匹配的条件序号

#处理完每个采样点，直到结束

while True:

#wait指定了匹配条件，第一次flag\_dex值为０，所以使用了第一个条件，即

#向上边沿

(a,b) = self.wait(flag\_arr[flag\_dex])

#刚才是向上边沿查找条件，接下来要切换到向下边沿查找条件

if flag\_dex == 0:

flag\_dex = 1

lst\_dex = self.samplenum　#保存向上边沿的采样位置

else:

flag\_dex = 0 #切换成向上边沿查找条件

times += 1

if times % 2 == 0:

#偶数次输出在第一行

self.put\_row1(lst\_dex, self.samplenum, self.samplenum - lst\_dex)

else:

#奇数次输出在第二行

self.put\_row2(lst\_dex, self.samplenum, self.samplenum - lst\_dex)

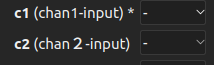


图１



图２

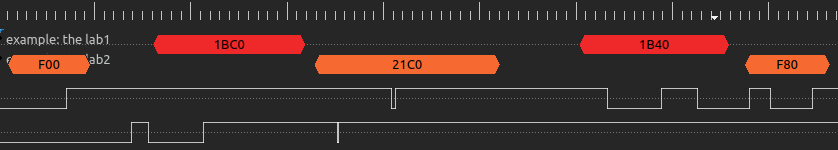


图３