

自定义协议解码器开发教程（以 my_uart 为例）

版本要求： 请确保当前上位机为最新版本，且版本号在 **V1.0.3 及以上**，否则自定义协议的加载、设置项或波形显示可能与本文描述不一致。

读者： 拿到 **decoders** 目录的二次开发者 / 普通程序员。

你能接触到的内容： 各协议文件夹里的 Python 文件（如 **my_uart/__init__.py**、**my_uart/pd.py**），以及软件界面上的「协议列表 / 设置窗 / 波形注解」。

你看不到、也不需要改： 上位机程序本身的源码。本文只从「界面上看见什么」说明联动关系。

目标： 读懂 **my_uart**，能改参数与显示，并能自己做一个新协议。

运行时软件会扫描：**可执行文件同级目录下的 **decoders/****。请把改好的协议文件夹放到该目录后重启软件。

目录

- 1. 先建立直觉：改 Python，界面怎么变
- 2. 一个协议文件夹长什么样
- 3. Python 基础教程（带示例）
- 4. 解码器专用 API（带示例）
- 5. 模块约定：**__init__.py** + **pd.py**
- 6. 逐块拆解 **my_uart**
- 7. Python ↔ 软件界面对照表
- 8. 动手练习 A：改现有协议
- 9. 动手练习 B：从零做一个 **mini_proto**
- 10. 部署、调试与常见坑
- 11. 附录：UART 帧与状态机速览

1. 先建立直觉：改 Python，界面怎么变

协议解码器 = 一个小 **Python** 插件。你写清楚三件事：

- 1. 需要哪些信号脚（通道）
- 2. 用户要填哪些参数（波特率、校验等）
- 3. 解码后在波形上画什么字

你在 pd.py 里改的东西	软件界面上你会看到
id / name	「添加协议」列表里多一个名字（如 MY_UART ）
channels / optional_channels	设置窗口多出通道下拉框（如 RX、TX）
options 增加一项	设置窗口多出一行数字框或下拉框
annotations + annotation_rows	波形下方多出一行（或一类）彩色注解条
put(...) / putgse(...) 写出的文字	注解条上的具体内容（如 0x55 、 Start ）

记住两句话：

- **类属性（元数据）** → 决定设置界面长什么样
- **运行时 put** → 决定波形上画什么内容

推荐操作路径（在软件里）：

1. 打开会话 → 添加协议 / 解码器
2. 在列表中选 **MY_UART**
3. 在设置窗绑定 RX/TX 到逻辑通道，改波特率等
4. 确定后，波形下方出现 RX data、TX data、bits 等注解行

本教程主要对照这两个目录（都在 **decoders** 下）：

目录	用途
my_uart/	教学用自定义 UART（带中文注释，推荐当模板）
uart/	官方 UART，可对照
i2c/	必选双线通道的例子（SCL/SDA）

2. 一个协议文件夹长什么样

decoders/
 my_uart/
 __init__.py
 pd.py
 uart/
 __init__.py
 pd.py
 i2c/
 ...

← 文件夹名必须等于下面的 **id**

← 导出 **Decoder**（可写模块注释供人阅读）

← 协议实现（你主要改这个）

软件启动时大致做这些事（你只需知道结果）：

扫描 **decoders** 下每个子目录

→ 读入 **__init__.py** / **pd.py**

→ 根据 **Decoder** 类上的「名片、通道、参数、注解行」生成设置界面

→ 用户点确定后，对采样数据调用 **decode()**

→ 你 **put** 出来的注解画在波形上

命名铁律：

项目	规则	例子
文件夹名	必须等于 id	文件夹 my_uart
id	小写 + 下划线，全局唯一	id = 'my_uart'
name	软件列表里显示的名字	name = 'MY_UART'

改完文件后：复制到软件旁的 **decoders/** → 重启软件，才能看到新协议或新参数。

3. Python 基础教程（带示例）

不必成为 Python 专家。下面每个知识点都配有「能在解码器里怎么用」的示例。

建议：先通读本节，再打开 **my_uart/pd.py** 对照。

3.1 注释

```
# 单行注释：从 # 一直到行尾

'''
多行注释 / 文档字符串
常写在文件顶部或函数下方，用来说明用途。
'''
```

在解码器中：**__init__.py** 顶部也可用三引号写模块说明，方便阅读源码的人理解协议；这与设置窗里的通道、参数、波形注解无关，改这里不会在软件界面上多出一项配置。

3.2 变量与基本类型

```
baud = 115200      # 整数 int
ratio = 1.5        # 浮点数 float
name = 'MY_UART'   # 字符串 str（单引号、双引号都可以）
ok = True          # 布尔 bool: True / False
empty = None       # 空值
```

解码器示例（在 **start()** 里把界面选项存成自己的变量）：

```
self.data_bits = self.options['data_bits']      # 可能是 8
self.show_start_stop = self.options['show_start_stop'] == 'yes' # 变成 True/False
```

3.3 缩进 = 代码块（最重要）

Python 不用 **{ }**，靠**缩进**表示「属于谁」。

```
def check(x):
    if x > 0:
        print('正数')      # 属于 if
        print('继续')      # 仍属于 if
    print('函数结尾')      # 属于 check，但不属于 if
```

错误示范（缩进乱了会直接报错，协议加载失败）：

```
def bad():  
    print('少缩进')    # IndentationError
```

规则：同一个文件里统一用 **4 个空格**，不要混用 Tab。

3.4 条件判断与比较

```
parity = 'odd'  
  
if parity == 'ignore':  
    print('不检查')  
elif parity == 'odd':  
    print('奇校验')  
else:  
    print('其它')  
  
# 常用比较  
# == 等于    != 不等于    < > <= >=  
# and 并且    or 或者    not 取反  
  
if baud > 0 and samplerate is not None:  
    bit_width = samplerate / baud
```

解码器示例（my_uart 里校验类型）：

```
if parity_type == 'zero':  
    return parity_bit == 0  
elif parity_type == 'one':  
    return parity_bit == 1
```

界面里选的是字符串 'yes' / 'no'，代码里常写成：

```
self.put_sample_points = self.options['put_sample_points'] == 'yes'  
# 选 yes → True; 选 no → False
```

3.5 循环

```
# for: 已知次数或遍历序列  
for i in range(8):          # i = 0,1,2,...,7  
    print(i)
```

```

for ch in ('rx', 'tx'):
    print(ch)

# while: 条件为真就一直转（解码主循环就是这种）
n = 0
while n < 3:
    n = n + 1

```

解码器示例：

```

# 为 RX、TX 两个方向各输出一个初始标记
for rxtx in (0, 1):
    self.putgse(0, 0, [Ann.RX_DATA + rxtx, ['标记']])

# decode() 主循环示意
while True:
    signal = self.wait(conds)  # 等到事件再继续
    # ... 处理 ...

```

`range(5)` → 0,1,2,3,4 (不含 5)。

`range(2, 6)` → 2,3,4,5。

3.6 列表 list（可变）

列表用方括号，可以增删改，适合缓存「正在收的那些位」。

```

bits = []                # 空列表
bits.append(1)           # [1]
bits.append(0)           # [1, 0]
bits.append(1)           # [1, 0, 1]

print(len(bits))         # 长度 3
print(bits[0])           # 第一个元素：1
print(bits[-1])          # 最后一个元素：1

bits.clear()             # 清空 → []

```

解码器示例：

```

self.databits = [[], []]  # 下标 0=RX, 1=TX 各一份列表
self.databits[0].append([1, ss, es])  # 记下：位值、起止采样点

```

3.7 元组 tuple（不可变）

元组用圆括号，**创建后不能改**。协议「名片」里大量用元组，因为加载后不该被改乱。

```
point = (100, 200)      # 两个数的元组
print(point[0])         # 100

# 只有一个元素时，要加逗号
single = (Ann.RX_DATA,) # 注意末尾逗号！
# wrong = (Ann.RX_DATA) # 这只是括号，不是元组
```

解码器示例：

```
optional_channels = (
    {'id': 'rx', 'name': 'RX', 'desc': 'UART 接收线'},
    {'id': 'tx', 'name': 'TX', 'desc': 'UART 发送线'},
)
```

3.8 字典 dict (键 → 值)

```
opt = {
    'id': 'baudrate',
    'desc': 'Baud rate(波特率)',
    'default': 115200,
}

print(opt['id'])      # 'baudrate'
print(opt['default']) # 115200

# 运行时用户选项也是字典
baud = self.options['baudrate']
```

小练习：下面这段和设置窗有什么关系？

```
{'id': 'data_bits', 'desc': 'Data bits(数据位数)', 'default': 8,
 'values': (5, 6, 7, 8, 9)}
```

答案：`id` 给代码用；`desc` 是设置窗左边的标签；`default` 是默认值；`values` 有则变成下拉框。

3.9 函数 def

```
def parity_ok(parity_type, parity_bit, data):
    '''校验是否正确。返回 True/False。'''
    if parity_type == 'ignore':
```

```

        return True
    ones = bin(data).count('1') + parity_bit
    if parity_type == 'odd':
        return (ones % 2) == 1
    return (ones % 2) == 0

# 调用
ok = parity_ok('even', 1, 0x55)

```

参数默认值:

```

def clamp(x, lo=0, hi=100):
    if x < lo:
        return lo
    if x > hi:
        return hi
    return x

clamp(120)          # → 100 (用了默认 hi)
clamp(120, hi=200)  # → 120

```

3.10 类 class (解码器就是一个类)

把「数据 + 方法」捆在一起。

```

class Counter:
    def __init__(self):          # 创建对象时自动调用
        self.n = 0              # self.xxx = 属于这个对象的数据

    def add(self, x):
        self.n = self.n + x

c = Counter()
c.add(3)
print(c.n)    # 3

```

继承: 在别人写好的类上扩展。

```

import sigrokdecode as srd

class Decoder(srd.Decoder):    # 继承框架基类
    id = 'my_uart'              # 类属性: 加载协议时读
    name = 'MY_UART'

```

```
def start(self):          # 实例方法：开始解码时调
    self.out_ann = self.register(srd.OUTPUT_ANN)
```

框架固定调用顺序（记住即可）：

```
创建 Decoder()
→ start()          # 读参数、注册输出
→ metadata(...)    # 告知采样率等
→ decode()         # 你的主循环
```

3.11 一次给多个常量编号：range 解包

```
# 手写很烦：
# RX_WARN = 0
# TX_WARN = 1
# RX_DATA = 2
# ...

# 一行搞定：
RX_WARN, TX_WARN, RX_DATA, TX_DATA = range(4)
# 得到：0, 1, 2, 3
```

my_uart 里真实写法：

```
class Ann:
    RX_WARN, TX_WARN, \
    RX_DATA, TX_DATA, \
    RX_START, TX_START, \
    ...
    FNIRSI_POINT = range(21)  # 自动得到 0~20 共 21 个编号
```

这些编号必须和后面的 annotations 下标一一对应（见第 6 章）。

3.12 字符串格式化（波形上要显示的字）

```
value = 85

# 方式 1: % (老式，解码器里很常见)
'0x%02X' % value      # → '0x55'    (两位十六进制，不足补 0)
'通道:%d' % 0          # → '通道:0'
'%d' % value          # → '85'

# 方式 2: format
```

```
'0x{:02X}'.format(value) # → '0x55'
'波特率={}'.format(115200)

# 方式 3: f-string (Python 3.6+)
baud = 115200
f'波特率={baud}'          # → '波特率=115200'
```

解码器示例：一次给出「长文本、短文本」，方便波形缩放时切换显示。

```
self.putgse(ss, es, [Ann.RX_DATA, ['0x55', '55', 'U']])
# 列表里：完整 → 较短 → 更短
```

3.13 导入模块 import

```
import sigrokdecode as srd          # 框架，必须
from common.srdhelper import bitpack # 公共小工具（把位列表拼成整数）

# 写在 __init__.py 里：把同目录 pd.py 里的 Decoder 暴露出去
from .pd import Decoder
```

bitpack 直观例子：

```
# 假设先收到的是 LSB：位列表 [1,0,1,0,1,0,1,0]
# bitpack 会按规则拼成整数（具体以 common 实现为准）
# 你在 get_data_bits 结束时 would 用到类似逻辑
```

3.14 异常：出错时主动告诉软件

```
class ChannelError(Exception):
    '''通道配置不对'''
    pass

# 使用
if not has_rx and not has_tx:
    raise ChannelError('需要至少配置一个通道：RX 或 TX')
```

软件会中止本次解码，并通常给出错误提示。调试时先检查：通道是否绑定、波特率是否为 0 等。

3.15 列表推导与简单推导（选学）

```
# 检查 RX、TX 是否都绑了通道
has_pin = [self.has_channel(ch) for ch in (0, 1)]
# 例如 → [True, False] 表示只绑了 RX

if not True in has_pin:
    raise ChannelError('至少一个通道')
```

等价于：

```
has_pin = []
for ch in (0, 1):
    has_pin.append(self.has_channel(ch))
```

3.16 小测验（建议自己在纸上写答案）

- 1. optional_channels 为什么用元组而不是列表？
 - 2. 'yes' == 'yes' 的结果是什么？ 'yes' == True 呢？
 - 3. (Ann.RX_DATA) 和 (Ann.RX_DATA,) 有什么区别？
 - 4. 为什么说 Ann.RX_DATA 的数值必须等于它在 annotations 里的下标？
- 参考答案：1) 元数据加载后不应被改。2) True；False（类型不同）。3) 前者只是整数，后者才是单元素元组。4) put 时用数字当「注解类型」，框架按这个数字归到对应行。

4. 解码器专用 API（带示例）

这些不是普通 Python 标准库，而是 sigrokdecode 框架提供的。写协议几乎总要用到。

4.1 注册输出通道 register

在 start() 里做一次：

```
def start(self):
    self.out_ann = self.register(srd.OUTPUT_ANN) # 波形注解（你最常用）
    self.out_python = self.register(srd.OUTPUT_PYTHON) # 给「上层堆叠协议」用
    self.out_binary = self.register(srd.OUTPUT_BINARY) # 二进制导出（本软件界面基本不用）
```

输出类型	作用	界面上
OUTPUT_ANN	画彩色注解条	看得见
OUTPUT_PYTHON	把结构化结果交给下一个协议	界面不直接画
OUTPUT_BINARY	导出原始字节	一般无界面

4.2 输出注解 put / putgse

```
# 标准写法
self.put(ss, es, self.out_ann, [注解类型编号, [长文本, 短文本, ...]])

# my_uart 封装后更短
self.putgse(ss, es, [Ann.RX_DATA, ['0x42', '66', 'B']])
```

参数	含义	界面效果
ss	起始采样点	色块左边界
es	结束采样点	色块右边界
注解类型编号	如 <code>Ann.RX_DATA</code>	落到哪一行、什么颜色类
文本列表	至少一个字符串	色块上显示的字

完整小示例：在采样点 1000~1100 标一个起始位。

```
ss, es = 1000, 1100
self.putgse(ss, es, [Ann.RX_START, ['Start', 'S']])
```

4.3 等待条件 wait（只能在 decode() 里）

```
def decode(self):
    while True:
        # 等到通道 0 出现下降沿
        pins = self.wait({0: 'f'})
        # 此时 self.samplenum 是当前采样序号
        self.put(self.samplenum, self.samplenum + 1,
                 self.out_ann, [0, ['FALL']])
```

常用边沿条件：

写法	含义
<code>{0: 'f'}</code>	通道 0 下降沿 falling
<code>{0: 'r'}</code>	通道 0 上升沿 rising
<code>{0: 'e'}</code>	通道 0 任意边沿 edge
<code>{0: 'h'} / {0: 'l'}</code>	保持高 / 低（配合超时等高级用法）

也可以一次等多个条件（`my_uart` 就是这样），然后用 `self.matched[i]` 判断是哪一个触发的。

硬规则：`wait()` **禁止** 写在普通子函数里，只能写在 `decode()`；否则可能崩溃。

4.4 读用户参数与通道

```
# 设置窗里用户选的值
baud = self.options['baudrate']
fmt = self.options['format']           # 'hex' / 'dec' / ...

# 用户是否绑定了某通道（下标与 channels/optional_channels 顺序一致）
if self.has_channel(0):
    print('RX 已绑定')
```

4.5 采样率元数据 metadata

```
def metadata(self, key, value):
    if key == srd.SRD_CONF_SAMPLERATE:
        self.samplerate = value
        self.baudrate = float(self.options['baudrate'])
        # 每一位占多少个采样点
        self.bit_width = self.samplerate / self.baudrate
```

示例：采样率 1 000 000 Hz，波特率 100 000 → 每位约 10 个采样点。

4.6 API 速查表

API	一句话
<code>self.register(...)</code>	注册输出
<code>self.put(ss, es, out, data)</code>	输出一条结果
<code>self.wait(conds)</code>	等到信号条件（仅 <code>decode</code> ）
<code>self.has_channel(i)</code>	第 <code>i</code> 路是否绑定
<code>self.options['id']</code>	读设置项
<code>self.sampenum</code>	当前采样点编号
<code>self.matched</code>	多个 <code>wait</code> 条件时，谁被触发了

5. 模块约定： `__init__.py` + `pd.py`

5.1 `__init__.py`

```
'''
MY_UART 协议解码器

UART 是一种简单的串行协议：
- 1 个起始位 (0)
- 5~9 个数据位
- 可选校验位
- 1 个或多个停止位 (1)
'''

from .pd import Decoder    # 必须：否则软件加载不到这个协议
```

内容	作用
顶部 <code>'''...'''</code> (可选)	给人看的模块说明，便于阅读源码； 不是 设置参数，也不决定波形显示
<code>from .pd import Decoder</code>	必须 ：没有这行 → 协议列表里可能根本没有该协议

5.2 pd.py 两层结构

类属性（元数据） <code>id, name, options, channels, annotations, annotation_rows ...</code>	→ 生成设置界面、注解行
实例方法（逻辑） <code>reset / start / metadata / decode</code>	→ 真正解码、put 文字

6. 逐块拆解 my_uart

文件：`decoders/my_uart/pd.py`（请边看边打开该文件）。

6.1 名片字段

```
api_version = 3
id = 'my_uart'
name = 'MY_UART'
longname = 'My Universal Asynchronous Receiver/Transmitter'
desc = '自定义UART协议解码器 - 学习示例版本'
inputs = ['logic']      # 直接吃逻辑通道 = 一级协议
outputs = ['uart']      # 输出类型名；其它协议若声明输入 uart，可叠在上面
```

字段	你在软件里的感受
name	列表显示 MY_UART
id	内部唯一 ID（须等于文件夹名）
inputs=['logic']	可以直接添加到会话，绑逻辑通道
outputs=['uart']	可被「基于 UART 的上层协议」继续解码
longname / desc	写在源码里的描述字段；设置窗主要认 name、通道、options 与注解行

6.2 通道：可选 vs 必选

my_uart 只有可选通道：

```
optional_channels = (  
    {'id': 'rx', 'name': 'RX', 'desc': 'UART 接收线 (RX 数据输入)'},  
    {'id': 'tx', 'name': 'TX', 'desc': 'UART 发送线 (TX 数据输出)'},  
)
```

对照 i2c（必选）：

```
channels = (  
    {'id': 'scl', 'name': 'SCL', 'desc': 'Serial clock line(串行时钟线)'},  
    {'id': 'sda', 'name': 'SDA', 'desc': 'Serial data line(串行数据线)'},  
)
```

写法	设置窗	用户操作
optional_channels	可选通道	可保持「-」不绑；但解码时至少要有一个有效脚
channels	必选通道	必须绑到具体逻辑通道，否则无法确定

name → 下拉左侧标签；desc → 旁边说明文字。

6.3 options：每一个都会变成设置行

```
options = (  
    {'id': 'baudrate', 'desc': 'Baud rate(波特率)', 'default': 115200},  
    {'id': 'data_bits', 'desc': 'Data bits(数据位数)', 'default': 8,  
      'values': (5, 6, 7, 8, 9)},  
    {'id': 'format', 'desc': 'Data format(数据格式)', 'default': 'hex',  
      'values': ('ascii', 'dec', 'hex', 'oct', 'bin')},  
    ...  
)
```

字段	界面表现
有 <code>values</code>	下拉框
无 <code>values</code> 、默认是数字	数字框
<code>desc</code>	左侧文字（建议中英都写一点，方便辨认）
<code>default</code>	第一次打开时的默认值
<code>id</code>	代码 <code>self.options['id']</code>

结论：在 `options` 里加一项 → 设置窗**自动多一行**，不用改别的程序。

6.4 `Ann` + `annotations` + `annotation_rows`（三件套）

这是最容易错、也最影响波形的一块。

① 编号

```
class Ann:
    RX_WARN, TX_WARN, RX_DATA, TX_DATA, ... = range(21)
```

② 类型表（下标 = `Ann` 值）

```
annotations = (
    ('rx-warning', 'RX warning'), # 下标 0 = Ann.RX_WARN
    ('tx-warning', 'TX warning'), # 1
    ('rx-data', 'RX data'),       # 2 = Ann.RX_DATA
    ...
)
```

③ 分行（决定波形有几条轨）

```
annotation_rows = (
    ('rx-data-vals', 'RX data',
     (Ann.RX_DATA, Ann.RX_START, Ann.RX_PARITY_OK, Ann.RX_PARITY_ERR,
      Ann.RX_STOP)),
    ('rx-warnings', 'RX warnings', (Ann.RX_WARN,)),
    ...
)
```

你做的修改	界面变化
<code>annotation_rows</code> 增加一行	波形下方 多一条 可开关的注解轨
<code>put</code> 使用某个 <code>Ann.xxx</code>	字出现在包含该类型的那一行

你做的修改

界面变化

三件套编号对不齐

字跑错行、错类，很难查

`fnirsi-signs` 一类内部标记行：用于数据点打点，行列表里可能不展示给用户开关。

6.5 生命周期四个方法

reset()

清空状态（重新解码前也会走）

start()

注册输出、读 `options`

metadata()

拿采样率，计算 `bit_width`

decode()

`while True: wait → 状态机 → put`

`my_uart` 状态机：

WAIT_FOR_START_BIT → GET_START_BIT → GET_DATA_BITS

→ GET_PARITY_BIT（可选）→ GET_STOP_BITS → 回到等待起始位

改「协议语义」主要改状态机；改「界面有什么选项/几行字」主要改元数据。

7. Python ↔ 软件界面对照表

7.1 设置窗口

Python	界面
<code>name</code>	协议标题
<code>channels</code>	必选通道下拉（必须选逻辑通道）
<code>optional_channels</code>	可选通道下拉（可 -）
<code>options</code>	波特率、校验等参数行
<code>annotation_rows</code>	有哪些注解行可显示

7.2 波形区域

Python	界面
<code>put / putgse</code> 的区间 <code>ss~es</code>	色块左右范围
注解类型 <code>Ann.xxx</code>	落到哪一行
文本列表	色块上的字
<code>put_sample_points=yes</code>	<code>samples</code> 行出现 0/1
<code>show_start_stop</code>	<code>data</code> 行是否标起停位

Python	界面
<code>show_data_point</code>	是否打数据点标记

7.3 改了也几乎看不见的（可先忽略）

- `binary` / 二进制导出相关输出
- 仅给堆叠协议用的 `OUTPUT_PYTHON`（界面不直接画）

8. 动手练习 A：改现有协议

操作步骤统一为：

1. 编辑 `decoders/my_uart/pd.py`（或你的工作副本）
2. 确保软件旁 `decoders/my_uart/` 已是最新文件
3. **重启软件**
4. 重新添加 MY_UART（旧会话可能还留着旧参数）

练习 A1：改默认波特率

把

```
{'id': 'baudrate', 'desc': 'Baud rate(波特率)', 'default': 115200},
```

改成 `'default': 9600`。

期望：设置窗打开时波特率默认是 9600。

练习 A2：加一个开关，并在波形上见字

在 `options` 末尾增加：

```
{'id': 'show_debug_tag', 'desc': 'Show debug tag(调试标记)', 'default': 'no',
 'values': ('yes', 'no')},
```

在 `start()` 中：

```
self.show_debug_tag = self.options['show_debug_tag'] == 'yes'
```

在输出一帧 RX 数据的附近增加：

```
if self.show_debug_tag:
    self.putgse(ss, es, [Ann.RX_WARN, ['DBG', 'D']])
```

期望：

- 设置窗多一行「Show debug tag...」下拉
- 选 yes 后，RX warnings 行出现 DBG

(这里故意复用已有 RX_WARN 行，改动最小。正规做法见 A3。)

练习 A3：正规新增一行注解

1. 把 Ann 的 range(21) 改成 range(22)，并增加常量 RX_DBG
2. annotations 末尾加 ('rx-debug', 'RX debug')
3. annotation_rows 增加 ('rx-debug', 'RX debug', (Ann.RX_DBG,))
4. putgse(..., [Ann.RX_DBG, ['HELLO']])

期望：波形多出一行 RX debug，上面显示 HELLO。

9. 动手练习 B：从零做一个 mini_proto

目标：检测某通道下降沿，在波形上标注文字。

9.1 建文件夹

```
decoders/mini_proto/
  __init__.py
  pd.py
```

9.2 __init__.py

```
'''
MINI_PROTO - 教学用最小解码器
检测所选通道的下降沿，并在波形上标注。
'''
from .pd import Decoder
```

9.3 pd.py

```
import sigrokdecode as srd

class Ann:
    FALL = 0

class Decoder(srd.Decoder):
    api_version = 3
    id = 'mini_proto'      # 必须 = 文件夹名
    name = 'MINI_PROTO'    # 列表里显示的名字
    longname = 'Minimal edge marker'
```

```

desc = '教学：下降沿标记'
license = 'gplv2+'
inputs = ['logic']
outputs = []
tags = ['Util']

channels = (
    {'id': 'din', 'name': 'DIN', 'desc': 'Input signal(输入信号)'},
)

options = (
    {'id': 'label', 'desc': 'Label text(标注文字)', 'default': 'FALL',
     'values': ('FALL', 'EDGE', '↓')},
)

annotations = (
    ('fall', 'Falling edge'),
)
annotation_rows = (
    ('edges', 'Edges', (Ann.FALL,)),
)

def __init__(self):
    self.reset()

def reset(self):
    pass

def start(self):
    self.out_ann = self.register(srd.OUTPUT_ANN)
    self.label = self.options['label']

def decode(self):
    while True:
        self.wait({0: 'f'})          # 通道 0 下降沿
        ss = self.samplenum
        self.put(ss, ss + 1, self.out_ann, [Ann.FALL, [self.label]])

```

9.4 验收清单

检查项	通过标准
列表	出现 MINI_PROTO
设置窗	有 DIN 通道 + Label 下拉
波形	Edges 行在下降沿处出现 FALL / EDGE / ↓

9.5 下一步怎么长成「像 my_uart」

需求	做法
要波特率	加 <code>options</code> ，在 <code>metadata</code> 里用采样率算位宽
要 RX/TX	用 <code>optional_channels + has_channel</code>
要多种注解	扩展 <code>Ann / annotations / annotation_rows</code>
给上层协议用	<code>outputs=['xxx']</code> ，并 <code>register(OUTPUT_PYTHON)</code>
叠在 UART 上	<code>inputs=['uart']</code> （二级协议，不再直接吃 logic）

10. 部署、调试与常见坑

10.1 部署

1. 修改 / 新建 `decoders/<协议名>/`
2. 放到正在运行的软件所使用的 `decoders` 目录（通常与可执行文件同级）
3. 重启软件
4. 重新添加协议实例（不要只依赖旧会话里的配置）

10.2 调试顺序

1. 列表里有没有？→ 查文件夹名、`id`、`from .pd import Decoder`、语法错误
2. 设置项对不对？→ 查 `options` / 通道定义
3. 波形有没有字？→ 查是否 `put`、`Ann` 编号、是否绑了通道
4. 字在错误的行？→ 查三件套是否对齐

10.3 常见坑

现象	可能原因
列表没有新协议	文件夹名 $\neq$ <code>id</code> ；文件没拷到软件旁； <code>__init__</code> 未导出；Python 语法错误
有协议但没有新参数	改错了另一份 <code>decoders</code> 副本；没重启；会话还是旧配置
字跑到奇怪的行	<code>Ann</code> 与 <code>annotations</code> 顺序不一致
解码时崩溃	在非 <code>decode()</code> 处调用了 <code>wait()</code>
无法确定协议	必选 <code>channels</code> 仍是「-」
改了 <code>binary</code> 无变化	界面基本不使用二进制输出

11. 附录：UART 帧与状态机速览

11.1 默认 8N1 一帧

空闲(高) | 起始(低) | D0..D7 | [校验] | 停止(高) | 空闲...

- 位宽（采样点数） $\approx$ 采样率 / 波特率

- 默认在位宽约 50% 处采样 (`sample_point`)
- 常用 LSB-first (先发低位)

11.2 `my_uart` 主要 options

id	含义	界面控件
<code>baudrate</code>	波特率	数字框
<code>data_bits</code>	5~9	下拉
<code>parity</code>	none/odd/even/...	下拉
<code>stop_bits</code>	0.5~2.0	下拉
<code>bit_order</code>	lsb/msb	下拉
<code>format</code>	显示进制	下拉
<code>invert_rx / invert_tx</code>	电平反相	下拉
<code>put_sample_points</code>	是否画采样点	下拉 → samples 行
<code>show_start_stop</code>	是否标起停位	下拉
<code>sample_point</code>	采样位置 %	数字框
<code>packet_*</code>	按空闲/分隔符/长度组包	影响 packets 行
<code>show_data_point</code>	数据点标记	下拉

11.3 推荐阅读 `my_uart` 的顺序

1. `__init__.py`
2. `pd.py` 顶部: `Ann`、`options`、`annotations`、`annotation_rows`
3. `start()`、`metadata()`
4. `decode()` 主循环
5. `get_start_bit` / `get_data_bits` 等处理函数
6. 有余力再对照 `uart/pd.py`、`i2c/pd.py`

修订记录

日期	说明
2026-08-06	初版

文档路径: `doc/自定义协议解码器开发教程-my_uart.md`。