PXView C 解码器测试指南

自动化测试系统：数据生成、并行运行与 C/Python 一致性校验 · PXLogic 官方文档 · fpgausb.com

1. 概述

PXView 项目包含 215 个原生 C 协议解码器（位于 `libsigrokdecode/c_decoders/`），每个解码器以 DLL 形式编译，由 `libsigrokdecode` 在运行时动态加载。为确保 C 解码器与对应 Python 解码器的行为一致性，项目提供了一套自动化测试系统：

- **测试工具**：`decoder_test.exe` — C 语言编写的测试驱动程序，加载解码器、输入逻辑数据、收集注释输出并输出 JSON
- **测试运行器**：`run_all_tests.py` — Python 编写的并行测试调度器，自动运行 C 和 Python 解码器并比较输出
- **数据生成器**：`generate_testdata.py` — 自动解析 C 解码器源文件，提取通道/选项元数据，并使用协议合成器生成测试数据
- **协议合成器**：`fuzzers/` — 包含各协议的 `BitstreamBuilder` 子类，可生成真实的协议时序波形

测试的核心思路：**对同一份输入数据，分别运行 C 解码器和 Python 解码器，比较两者输出的注释（annotation）是否一致。**

2. 测试数据生成流程

2.1 自动生成

使用 `generate_testdata.py` 可自动为所有 C 解码器生成测试数据：

```
#
python libsigrokdecode/tests/generate_testdata.py

#
python libsigrokdecode/tests/generate_testdata.py --overwrite

#    C
python libsigrokdecode/tests/generate_testdata.py \
    --c-decoders-dir libsigrokdecode/c_decoders \
    --output-dir libsigrokdecode/tests/testdata
```

脚本工作流程：

扫描 `c_decoders/` 目录下所有 `*_c.c` 文件

解析源文件，提取通道定义、选项定义、输入类型等元数据

对于有逻辑输入的解码器，尝试使用 `fuzzers/` 中的协议合成器生成真实协议数据；若无对应合成器，则生成简单的模式数据

对于非逻辑输入的解码器（即输入来自其他解码器），生成带 `needs_upstream` 标记的配置

对于堆叠解码器（如 `ethernet_c` 需要 `nrzi_c → 4b5b_c` 链），使用 `_STACK_INPUT_OVERRIDES` 中预定义的配置

输出 `config.json` 和 `input.bin` 到 `testdata/<decoder_name>/default/` 目录

2.2 手动创建测试数据

测试数据目录结构：

```
testdata/
└── /
    └── default/          #          "default"
        ├── config.json   #
        └── input.bin     #
```

config.json 格式

```
{
    "decoder": "<   ID>",
    "samplerate": <   (Hz)>,
    "num_channels": <   >,
    "sample_count": <    >,
    "channels": {
        "<  ID>": <    >,
        ...
    },
    "options": {
        "<  ID>": <   >,
        ...
    }
}
```

各字段说明：

字段	类型	必填	说明
decoder	string	是	解码器 ID，如 "spi_c" 、"i2c_c"
samplerate	integer	是	采样率，单位 Hz
num_channels	integer	是	输入逻辑通道总数
sample_count	integer	是	总采样点数
channels	object	是	通道 ID 到通道索引的映射
options	object	否	解码器选项（使用默认值时可省略）
stack	array	否	堆叠解码器配置（见下文）
needs_upstream	boolean	否	标记该解码器需要上游输出，测试时跳过
expected_deviations	boolean	否	标记已知偏差，测试时降级为 DEVIATION

input.bin 格式

`input.bin` 采用**位打包（bit-packed）**格式存储逻辑信号数据：

- 每个通道的数据独立存储，按通道索引顺序排列
- 每个通道占用 `ceil(sample_count / 8)` 字节
- 位序：**LSB-first**，即 byte 0 的 bit 0 = sample 0，bit 1 = sample 1，以此类推
- 总文件大小 = `num_channels × ceil(sample_count / 8)` 字节

```
┌──────────────────┬──────────────────┬─────┬──────────────────┐
│   Channel 0      │   Channel 1      │ ... │  Channel N-1     │
│  (bytes_per_ch)  │  (bytes_per_ch)  │     │  (bytes_per_ch)  │
└──────────────────┴──────────────────┴─────┴──────────────────┘


Byte 0: [S7 S6 S5 S4 S3 S2 S1 S0]   ← S0   sample 0 LSB
Byte 1: [S15 S14 S13 S12 S11 S10 S9 S8]
...
```

简单解码器配置示例（counter_c）

```
{
    "decoder": "counter_c",
    "samplerate": 1000000,
    "num_channels": 2,
    "sample_count": 10000,
    "channels": {
        "data": 0,
        "reset": 1
    },
    "options": {
        "data_edge": "any",
        "divider": 0,
        "reset_edge": "falling",
        "edge_off": 0,
        "word_off": 0,
        "dead_cycles": 0,
        "start_with_reset": "no"
    }
}
```

堆叠解码器配置示例（ethernet_c）

堆叠解码器是指输入来自其他解码器输出（而非原始逻辑信号）的解码器。通过 `stack` 字段定义解码器链：

```
{
    "decoder": "ethernet_c",
    "samplerate": 1000000,
    "num_channels": 1,
    "sample_count": 100000,
    "channels": {},
    "stack": [
        {
            "id": "nrzi_c",
            "channels": {
                "data": 0
            }
        },
        {
            "id": "4b5b_c"
        }
    ]
}
```

`stack` 数组定义了解码器链的底层部分，数据流为：

```
input.bin → nrzi_c → 4b5b_c → ethernet_c
```

- `stack` 中第一个元素必须指定 `channels`（映射到 `input.bin` 的通道）
- 后续元素可省略 `channels`，系统会自动按顺序映射

- 顶层解码器的 `channels` 通常为空 `{}`，因为其输入来自堆叠链的上一级

3. 单解码器验证流程

步骤 1：构建项目

确保 C 解码器 DLL 已编译：

```
build_incremental.cmd
```

编译完成后，DLL 位于 `build.dir/decoders/c_decoders/` 目录，`decoder_test.exe` 位于 `build.dir/` 或 `build/bin/` 目录。

步骤 2：运行单个解码器测试

```
cd libsigrokdecode/tests
python run_all_tests.py --decoder spi_c
```

此命令会查找 `testdata/spi_c/` 下所有子目录（测试场景），分别运行 C 和 Python 解码器并比较结果。

步骤 3：仅生成 C 解码器输出

```
decoder_test.exe -d spi_c -t testdata/spi_c/default -f actual_c.json --generate-only
```

参数说明：

参数	说明
-d	解码器 ID（C 解码器使用带 _c 后缀的名称）
-t	测试数据目录（包含 config.json 和 input.bin ）
-f	输出 JSON 文件路径
--generate-only	仅运行解码器并输出结果，不与 expected.json 比较
--tolerance N	采样点比较容差（默认 0，通常设为 2）

步骤 4：仅生成 Python 解码器参考输出

```
decoder_test.exe -d spi -t testdata/spi_c/default -f expected_py.json --python --generate-only
```

注意：Python 模式下解码器 ID 不带 `_c` 后缀。`--python` 标志使 `decoder_test.exe` 加载 Python 解码器而非 C 解码器。

步骤 5：手动比较输出

```
#    diff
diff actual_c.json expected_py.json

#     Python
python -c "
import json
with open('actual_c.json') as f: c = json.load(f)
with open('expected_py.json') as f: p = json.load(f)
print(f'C annotations: {len(c.get(\"annotations\", []))}')
print(f'Py annotations: {len(p.get(\"annotations\", []))}')
"
```

输出 JSON 格式：

```
{
    "decoder": "spi_c",
    "samplerate": 1000000,
    "num_annotations": 12,
    "annotations": [
        {
            "start_sample": 100,
            "end_sample": 200,
            "ann_class": 0,
            "ann_type": 1,
            "texts": ["0xDE", "11011110"]
        }
    ]
}
```

步骤 6：运行全部测试

```
python run_all_tests.py --all

#          16
python run_all_tests.py --all --jobs 8
```

测试完成后会生成： - `test_results.csv` — 详细的测试结果表格 - `Dashboard.md` — Markdown 格式的测试仪表盘，按状态优先级排序

4. 测试结果解读

测试系统定义了 6 种状态：

PASS □

C 和 Python 解码器输出完全匹配。匹配规则： - 注释按 `(start_sample, ann_class)` 分组 - `end_sample` 允许 ±2 采样点容差 - 文本比较支持语义数值比较（见下文）

DEVIATION □

C 和 Python 输出存在偏差，但 `config.json` 中标记了 `expected_deviations: true`。这表示该偏差是已知的、可接受的差异（如 C 解码器实现了 Python 解码器未有的功能）。

```
{
    "decoder": "some_decoder_c",
    "expected_deviations": true,
    ...
}
```

WARN □

C 和 Python 解码器均输出 0 条注释。虽然形式上"匹配"，但这种空匹配没有实际意义，可能表示： - 输入数据不包含有效协议帧 - 解码器配置有误 - 采样率不足

FAIL □

C 和 Python 的注释输出不匹配。可能原因： - 注释数量不同 - `start_sample` / `end_sample` 超出容差 - `ann_class` 或 `ann_type` 不同 - 注释文本内容不匹配

失败时会输出详细的偏差报告，例如：

```
3 matches, 2 deviations found.
MISSED at sample 100: Py has class 0 (START) but C doesn't
EXTRA at sample 200: C has class 1 (DATA) but Py doesn't
```

ERROR □

解码器运行时崩溃或超时（默认 30 秒）。可能原因： - 解码器 DLL 加载失败 - 解码器内部段错误 - Python 解码器导入错误 - 输入数据格式错误

SKIP □

`config.json` 中标记了 `needs_upstream: true`，表示该解码器需要上游解码器的输出作为输入，无法直接用逻辑数据测试。

5. 自定义测试数据创建指南

5.1 使用 Python 脚本生成 input.bin

以下是一个完整的示例，展示如何为 SPI 解码器生成测试数据：

```python
#!/usr/bin/env python3
"""Generate test data for SPI decoder."""

import json
import math
import os

class BitstreamBuilder:
    def __init__(self, num_channels, sample_count, samplerate=1000000):
        self.num_channels = num_channels
        self.sample_count = sample_count
        self.samplerate = samplerate
        self.channels = [[0] * sample_count for _ in range(num_channels)]
        self.pos = 0

    def set_level(self, ch, level, duration_samples=1):
        for i in range(duration_samples):
            if self.pos + i < self.sample_count:
                self.channels[ch][self.pos + i] = 1 if level else 0
        self.pos += duration_samples

    def set_idle(self, ch, level):
        for i in range(self.pos, self.sample_count):
            self.channels[ch][i] = 1 if level else 0

    def get_bitpacked(self):
        result = bytearray()
        bytes_per_channel = math.ceil(self.sample_count / 8)
        for ch in range(self.num_channels):
            packed = bytearray(bytes_per_channel)
            for i, val in enumerate(self.channels[ch]):
                if val:
                    packed[i // 8] |= (1 << (i % 8))
            result.extend(packed)
        return bytes(result)


class SPIGenerator:
```

```
    def __init__(self, builder, clk_ch, mosi_ch, miso_ch, cs_ch):
        self.builder = builder
        self.clk = clk_ch
        self.mosi = mosi_ch
        self.miso = miso_ch
        self.cs = cs_ch

    def select(self):
        self.builder.set_level(self.cs, 0, 0)  # CS active-low

    def deselect(self):
        self.builder.set_level(self.cs, 1, 0)  # CS inactive

    def write_byte(self, byte_val):
        for bit in range(7, -1, -1):  # MSB first
            level = (byte_val >> bit) & 1
            self.builder.set_level(self.mosi, level, 0)
            self.builder.set_level(self.clk, 1, 5)  # CLK high for 5 samples
            self.builder.set_level(self.clk, 0, 5)  # CLK low for 5 samples


def main():
    samplerate = 1000000   # 1 MHz
    sample_count = 10000
    num_channels = 4       # CLK, MOSI, MISO, CS

    builder = BitstreamBuilder(num_channels, sample_count, samplerate)
    builder.pos = 200  # Start offset

    # Channel mapping: clk=0, mosi=1, miso=2, cs=3
    gen = SPIGenerator(builder, clk_ch=0, mosi_ch=1, miso_ch=2, cs_ch=3)

    # CS idle high
    builder.set_idle(3, 1)
    builder.set_level(3, 1, builder.pos)

    # Transaction: write 0xDE then 0xAD
    gen.select()
    gen.write_byte(0xDE)
    gen.write_byte(0xAD)
    gen.deselect()

    # Set idle states for remaining samples
    builder.set_idle(0, 0)  # CLK idle low
    builder.set_idle(1, 0)  # MOSI idle low
    builder.set_idle(3, 1)  # CS idle high

    # Write config.json
    config = {
        "decoder": "spi_c",
        "samplerate": samplerate,
        "num_channels": num_channels,
        "sample_count": sample_count,
        "channels": {
            "clk": 0,
            "mosi": 1,
            "miso": 2,
            "cs": 3
        },
        "options": {
```

```
            "cs_polarity": "active-low",
            "cpol": 0,
            "cpha": 0,
            "bitorder": "msb-first",
            "wordsize": 8,
            "format": "hex",
            "show_data_point": "yes"
        }
    }

    output_dir = "testdata/spi_c/custom"
    os.makedirs(output_dir, exist_ok=True)

    with open(os.path.join(output_dir, "config.json"), "w") as f:
        json.dump(config, f, indent=2, ensure_ascii=False)
        f.write("\n")

    with open(os.path.join(output_dir, "input.bin"), "wb") as f:
        f.write(builder.get_bitpacked())

    print(f"Generated test data in {output_dir}")
    print(f"  config.json: {len(json.dumps(config))} bytes")
    print(f"  input.bin: {len(builder.get_bitpacked())} bytes")


if __name__ == "__main__":
    main()
```

5.2 UART 测试数据生成示例

```
#!/usr/bin/env python3
"""Generate test data for UART decoder."""

import json
import math
import os


class BitstreamBuilder:
    def __init__(self, num_channels, sample_count, samplerate=1000000):
        self.num_channels = num_channels
        self.sample_count = sample_count
        self.samplerate = samplerate
        self.channels = [[0] * sample_count for _ in range(num_channels)]
        self.pos = 0

    def set_level(self, ch, level, duration_samples=1):
        for i in range(duration_samples):
            if self.pos + i < self.sample_count:
                self.channels[ch][self.pos + i] = 1 if level else 0
        self.pos += duration_samples

    def set_idle(self, ch, level):
        for i in range(self.pos, self.sample_count):
            self.channels[ch][i] = 1 if level else 0

    def get_bitpacked(self):
        result = bytearray()
        bytes_per_channel = math.ceil(self.sample_count / 8)
        for ch in range(self.num_channels):
```

```
            packed = bytearray(bytes_per_channel)
            for i, val in enumerate(self.channels[ch]):
                if val:
                    packed[i // 8] |= (1 << (i % 8))
            result.extend(packed)
        return bytes(result)


class UARTGenerator:
    def __init__(self, builder, rx_ch, baudrate=115200):
        self.builder = builder
        self.rx = rx_ch
        self.baudrate = baudrate
        self.samples_per_bit = builder.samplerate // baudrate

    def write_byte(self, byte_val):
        # Start bit (0)
        self.builder.set_level(self.rx, 0, self.samples_per_bit)
        # Data bits (LSB first)
        for bit in range(8):
            level = (byte_val >> bit) & 1
            self.builder.set_level(self.rx, level, self.samples_per_bit)
        # Stop bit (1)
        self.builder.set_level(self.rx, 1, self.samples_per_bit)


def main():
    samplerate = 1000000   # 1 MHz
    baudrate = 115200
    sample_count = 10000
    num_channels = 2       # RX, TX

    builder = BitstreamBuilder(num_channels, sample_count, samplerate)
    builder.pos = 500  # Start offset

    # RX idle high (UART idle state)
    builder.set_idle(0, 1)
    builder.set_level(0, 1, builder.pos)

    # TX idle high
    builder.set_idle(1, 1)
    builder.set_level(1, 1, builder.pos)

    # Generate RX data: send 0x55, 0xAA, 0x12
    gen = UARTGenerator(builder, rx_ch=0, baudrate=baudrate)
    gen.write_byte(0x55)
    gen.write_byte(0xAA)
    gen.write_byte(0x12)

    # Set idle for remaining samples
    builder.set_idle(0, 1)
    builder.set_idle(1, 1)

    config = {
        "decoder": "uart_c",
        "samplerate": samplerate,
        "num_channels": num_channels,
        "sample_count": sample_count,
        "channels": {
            "rx": 0,
```

```
            "tx": 1
        },
        "options": {
            "baudrate": baudrate,
            "data_bits": 8,
            "stop_bits": 1.0,
            "parity": "none",
            "bit_order": "lsb-first",
            "format": "hex"
        }
    }

    output_dir = "testdata/uart_c/custom"
    os.makedirs(output_dir, exist_ok=True)

    with open(os.path.join(output_dir, "config.json"), "w") as f:
        json.dump(config, f, indent=2, ensure_ascii=False)
        f.write("\n")

    with open(os.path.join(output_dir, "input.bin"), "wb") as f:
        f.write(builder.get_bitpacked())

    print(f"Generated UART test data in {output_dir}")
    print(f"  Samples per bit: {samplerate // baudrate}")


if __name__ == "__main__":
    main()
```

5.3 关键注意事项

采样率选择：采样率必须足够高，确保每个协议比特有足够的采样点。一般规则： - UART：至少 4× 波特率（推荐 8× 以上） - SPI/I2C：至少 4× 时钟频率 - 高速协议（USB、CAN-FD）：需要 24MHz 以上

通道映射：`channels` 中的通道索引必须与 `input.bin` 中的通道顺序一致。通道索引从 0 开始。

空闲状态：生成数据前设置正确的空闲电平（如 UART 空闲为高，SPI CS 空闲为高），并在数据结束后用 `set_idle()` 填充剩余采样点。

起始偏移：建议在数据起始前留出少量空闲采样（如 200-2000 个），让解码器有时间检测空闲状态。

堆叠解码器：若需测试堆叠解码器，在 `config.json` 中添加 `stack` 字段，并确保 `input.bin` 包含底层解码器所需的全部通道数据。

6. 测试框架内部机制

6.1 并行执行

`run_all_tests.py` 使用 `concurrent.futures.ThreadPoolExecutor` 实现并行测试：

```
with ThreadPoolExecutor(max_workers=args.jobs) as executor:
    f_to_test = {executor.submit(run_test, d, p): (d, p) for d, p in test_cases}
    for i, f in enumerate(as_completed(f_to_test), 1):
        status, detail, elapsed = f.result()
```

默认并行度为 16，可通过 `--jobs` 参数调整。每个测试用例在独立线程中运行，互不干扰。

6.2 解码器 ID 映射

C 解码器 ID 与 Python 解码器 ID 的映射规则：

去除 `_c` 后缀（如 `spi_c` → `spi`）

查找硬编码映射表 `HARDCODED_ID_MAP`（处理无法通过规则推导的 ID）

Python 模式下，还会尝试多种变体（下划线/连字符替换、大小写变换等）

硬编码映射示例：

```
HARDCODED_ID_MAP = {
    'cjtag_oscan0': 'cjtag_oscan1',
    'eth_an': 'eth_auto_negotiation',
    'qspi': 'smart_qspi',
    'onewire': 'onewire_link',
    'delta-sigma': 'delta_sigma',
}
```

6.3 注释分组与匹配

比较算法的核心步骤：

分组：将 C 和 Python 的注释分别按 `(start_sample, ann_class)` 键分组

贪心匹配：对每个分组键，使用贪心算法进行 1-to-1 匹配 - 遍历 Python 注释列表，为每个 Python 注释寻找最佳匹配的 C 注释 - 匹配条件：`end_sample` 差值 ≤ 2 且所有文本语义相等 - 匹配成功则标记两个注释为已使用

偏差报告：未匹配的 Python 注释标记为 `MISSED`，未匹配的 C 注释标记为 `EXTRA`

6.4 语义文本比较

文本比较不仅支持精确匹配，还支持语义数值比较：

```
def compare_text_semantic(py_text, c_text):
    # 1.
    if py_text == c_text:
        return True

    # 2.   +
    #      "  +  "      "1.5ms", "100kHz"
    py_vals = parse_all_numerics_with_units(py_text)
    c_vals = parse_all_numerics_with_units(c_text)

    #                 1%
    if len(py_vals) > 0 and len(py_vals) == len(c_vals):
        for pv, cv in zip(py_vals, c_vals):
            rel_diff = abs(pv - cv) / max(abs(pv), abs(cv), 1e-12)
            if rel_diff > 0.01:
                return False
        return True

    # 3.                  μ/u/μ
    def normalize(t):
        return t.replace(' ', '').replace('μ', 'μ').replace('u', 'μ')
    return normalize(py_text) == normalize(c_text)
```

支持的单位：`ns`、`µs`、`ms`、`s`、`Hz`、`kHz`、`MHz`、`GHz`、`%`。

6.5 Dashboard 生成

测试完成后自动生成 `Dashboard.md`，包含：

- **Summary 表格**：各状态计数

- **Failures & Errors 详情**：失败和错误的具体信息
- **All Decoders 表格**：所有解码器的测试结果，按状态优先级排序（FAIL > ERROR > WARN > DEVIATION > PASS > SKIP）

6.6 decoder_test.exe 工作流程

`decoder_test.exe` 是 C 语言编写的测试驱动程序，其核心流程：

读取 `config.json` 获取解码器配置

读取 `input.bin` 并按通道拆分为独立的位打包数组

初始化 `libsigrokdecode`（C 模式加载 DLL，Python 模式加载 Python 解码器）

创建解码器实例，设置通道映射和选项

如有 `stack` 配置，创建堆叠解码器链

注册注释收集回调

启动会话，发送采样数据

收集所有注释输出，转换为 JSON 格式

若非 `--generate-only` 模式，与 `expected.json` 比较并输出 PASS/FAIL

退出码：0 = PASS，1 = FAIL，2 = ERROR。