PXView C 协议解码器开发指南

C 解码器 API v4 完整参考 · PXLogic 官方文档 · fpgausb.com

1. 概述

PXView 的协议解码引擎 `libsigrokdecode` 支持两种解码器实现方式：Python 解码器和 C 解码器。C 解码器以独立 DLL 形式编译，当前项目包含 **215 个** C 解码器，覆盖 SPI、I²C、UART、CAN、JTAG 等常见协议。

C 解码器核心特性

- **API 版本**：v4（`SRD_C_DECODER_API_VERSION = 4`）
- **编译方式**：每个解码器编译为独立的 `MODULE` DLL，运行时由 `libsigrokdecode` 动态加载
- **C□C 堆叠**：C 解码器之间可通过 `c_proto()` / `decode_upper` 机制堆叠，独立于 Python 解码器体系
- **性能**：相比 Python 解码器，C 解码器无需 Python 解释器开销，解码速度显著提升
- **兼容性**：C 解码器的通道定义、注释类编号、协议命令字符串必须与对应 Python 解码器完全一致，以确保前端 UI 和堆叠解码器的兼容性

术语对照

C 术语	Python 等价	说明
c_wait()	self.wait()	等待采样条件
c_put()	self.put()	输出注释
c_proto()	self.putp()	输出协议数据
c_pin()	self.samplenum 时刻的引脚值	读取引脚电平
di_samplenum()	self.samplenum	当前采样号
di_matched()	self.matched	条件匹配位掩码
decode_upper	self.wait() + self.decode()	接收上层协议数据

2. API v4 完整参考

2.1 `c_wait()` — 变参声明式条件等待

```
int c_wait(struct srd_decoder_inst *di, ...);
```

语义：暂停解码，直到指定的采样条件满足。参数为变参列表，由 `CW_*` 宏构建条件，以 `CW_END` 终止。返回 `SRD_OK`（0）表示条件满足，非零表示解码应终止。

Python 等价：`self.wait([{ch0: 'r'}, {ch0: 'h', ch1: 'f'}])`

条件宏：

宏	编码	Python 等价	含义
CW_H(ch)	HIGH	'h'	通道 ch 为高电平

宏	编码	Python 等价	含义
CW_L(ch)	'l'	通道 ch 为低电平	
CW_R(ch)	RISING_EDGE	'r'	通道 ch 上升沿
CW_F(ch)	FALLING_EDGE	'f'	通道 ch 下降沿
CW_E(ch)	EITHER_EDGE	'e'	通道 ch 任意边沿
CW_N(ch)	NO_EDGE	'n'	通道 ch 无边沿
CW_SKIP(n)	SKIP	{'skip': n}	跳过 n 个采样
CW_OR	—	列表中的 OR 分隔	分隔 OR 组
CW_END	—	—	终止条件列表

短别名（仅 C 语言可用，C++ 中不可用）：

短名	等价
H(ch)	CW_H(ch)
L(ch)	CW_L(ch)
R(ch)	CW_R(ch)
F(ch)	CW_F(ch)
E(ch)	CW_E(ch)
N(ch)	CW_N(ch)
SKIP(n)	CW_SKIP(n)
OR	CW_OR
END	CW_END

示例：

```
// Python: self.wait({0: 'r'})
c_wait(di, CW_R(0), CW_END);

// Python: self.wait([{0: 'r'}, {0: 'h', 1: 'f'}])
//    SCL    OR (SCL  AND SDA   )
c_wait(di, CW_R(0), CW_OR, CW_H(0), CW_F(1), CW_END);

// Python: self.wait({'skip': 100})
c_wait(di, CW_SKIP(100), CW_END);

// Python: self.wait()  (         )
c_wait(di, CW_END);
```

2.2 `c_put()` / `c_put_v()` / `c_put_t()` — 注释输出

```
#define c_put(di, ss, es, out_id, cls, ...)
#define c_put_v(di, ss, es, out_id, cls, val, ...)
#define c_put_t(di, ss, es, out_id, cls, tp, ...)
```

语义：向输出流发送注释（annotation）。参数为变参字符串列表，对应 Python 中的多级文本（长/中/短）。

函数	Python 等价	说明
c_put(di, ss, es, out, cls, ...)	self.put(ss, es, out, [cls, [texts]])	基本注释输出
c_put_v(di, ss, es, out, cls, val, ...)	同上 + 数值	带数值的注释，自动生成十六进制字符串
c_put_t(di, ss, es, out, cls, tp, ...)	同上 + 类型	带注释类型的输出

参数说明：

参数	类型	说明
di	struct srd_decoder_inst *	解码器实例
ss	uint64_t	起始采样号
es	uint64_t	结束采样号
out_id	int	输出 ID（由 c_reg_out() 返回）
cls	int	注释类编号（对应 ann_labels 数组下标）
val	整数	数值（c_put_v 专用，自动生成 0x%X 格式十六进制字符串）
tp	int	注释类型（c_put_t 专用）
...	字符串	变参文本列表，对应 Python 的 ['Long', 'Mid', 'Short']

示例：

```
// Python: self.put(ss, es, self.out_ann, [0, ['Start', 'S']])
c_put(di, ss, es, out_ann, ANN_START, "Start", "S");

// Python: self.put(ss, es, self.out_ann, [6, ['Address write', 'AW', '0x50']])
c_put(di, ss, es, out_ann, ANN_ADDRESS_WRITE, "Address write", "AW", "0x50");

// Python: self.put(ss, es, self.out_ann, [8, d])     d
c_put_v(di, ss, es, out_ann, ANN_DATA_READ, d, "Data read: 0x42", "DR: 42", "42");
```

2.3 `c_proto()` / `C_END` / `C_U8..C_BYTES` — 协议输出

```
int c_proto(struct srd_decoder_inst *di, uint64_t start_sample,
            uint64_t end_sample, int output_id,
            const char *cmd, ...);  /* C_END     c_field    */
```

语义：向堆叠的上层解码器发送协议数据。`cmd` 为命令字符串（如 `"ADDRESS WRITE"`），后续为 `c_field` 类型的字段参数，必须以 `C_END` 终止。

Python 等价：`self.putp(ss, es, cmd, field1, field2, ...)`

`c_field` **类型构造宏**：

宏	C 类型	Python 等价	说明
C_U8(v)	uint8_t	int	8 位无符号整数
C_U16(v)	uint16_t	int	16 位无符号整数
C_U32(v)	uint32_t	int	32 位无符号整数

宏	C 类型	Python 等价	说明
C_U64(v)	uint64_t	int	64 位无符号整数
C_I8(v)	int8_t	int	8 位有符号整数
C_I16(v)	int16_t	int	16 位有符号整数
C_I32(v)	int32_t	int	32 位有符号整数
C_I64(v)	int64_t	int	64 位有符号整数
C_F64(v)	double	float	64 位浮点数
C_STR(v)	const char *	str	字符串
C_BYTES(d, n)	const uint8_t *, uint32_t	bytes	字节数组
C_END	哨兵	—	终止 c_proto 参数列表

示例：

```
// Python: self.putp(ss, es, "ADDRESS WRITE", d)
c_proto(di, ss, es, out_proto, "ADDRESS WRITE", C_U8(d), C_END);

// Python: self.putp(ss, es, "DATA READ", data_byte)
c_proto(di, ss, es, out_proto, "DATA READ", C_U8(data_byte), C_END);

// Python: self.putp(ss, es, "BITS", bits_bytes)
c_proto(di, ss, es, out_proto, "BITS", C_BYTES(bits_data, bpos), C_END);

//
c_proto(di, ss, es, out_proto, "START", C_END);
c_proto(di, ss, es, out_proto, "STOP", C_END);
```

2.4 `c_pin()` / `di_samplenum()` / `di_matched()` — 快速访问

```
uint8_t c_pin(struct srd_decoder_inst *di, int ch);
```

语义：读取通道 `ch` 在当前采样点的电平值（0 或 1）。仅在 `c_wait()` 返回后有效。通道未连接时返回 `0xFF`。

Python 等价：`self.wait(...)` 后通过 `self.samplenum` 时刻的引脚值访问。

```
//    SDA
int sda_val = c_pin(di, SDA);

#define di_samplenum(di)   ((di)->abs_cur_samplenum)
```

语义：获取当前采样号。仅在 `c_wait()` 返回后有效。

Python 等价：`self.samplenum`

```
uint64_t samplenum = di_samplenum(di);

#define di_matched(di)     ((di)->match_array)
```

语义：获取条件匹配位掩码（`uint64_t`）。每一位对应 `c_wait()` 中的一个 OR 组。仅在 `c_wait()` 返回后有效。

Python 等价：`self.matched`（Python 中为 tuple of bool，C 中为整数位掩码）

```
// Python: if self.matched[0]:
// C:     0   OR
if (di_matched(di) & (1ULL << 0)) { ... }
```

```
// Python: if self.matched[1]:
// C:     1   OR
if (di_matched(di) & (1ULL << 1)) { ... }
```

2.5 `c_opt_int/str/dbl/bool()` — 选项读取

```
int64_t  c_opt_int(struct srd_decoder_inst *di, const char *key, int64_t defval);
const char *c_opt_str(struct srd_decoder_inst *di, const char *key, const char *defval);
double   c_opt_dbl(struct srd_decoder_inst *di, const char *key, double defval);
int      c_opt_bool(struct srd_decoder_inst *di, const char *key, int defval);
```

语义：读取解码器选项值。`key` 为选项 ID 字符串，`defval` 为选项不存在时的默认值。

Python 等价：`self.options[key]`

C 函数	Python 等价	返回类型
c_opt_int(di, "divider", 0)	self.options['divider']	int64_t
c_opt_str(di, "edge", "any")	self.options['edge']	const char *
c_opt_dbl(di, "threshold", 1.5)	self.options['threshold']	double
c_opt_bool(di, "invert", 0)	self.options['invert'] == 'yes'	int (0/1)

`c_opt_bool` **特殊说明**：接受 `"yes"/"true"/"1"` 为真，`"no"/"false"/"0"` 为假，其他值返回 `defval`。

示例：

```
int64_t divider = c_opt_int(di, "divider", 0);
const char *edge = c_opt_str(di, "data_edge", "any");
int show_data = c_opt_bool(di, "show_data_point", 1);
double threshold = c_opt_dbl(di, "threshold", 1.5);
```

2.6 `c_has_ch()` / `c_samplerate()` / `c_last_samplenum()` / `c_init_pin()` — 辅助函数

```
int      c_has_ch(struct srd_decoder_inst *di, int ch);
uint64_t c_samplerate(struct srd_decoder_inst *di);
uint64_t c_last_samplenum(struct srd_decoder_inst *di);
uint8_t  c_init_pin(struct srd_decoder_inst *di, int ch);
```

函数	Python 等价	说明
c_has_ch(di, ch)	self.has_channel(ch)	检查通道是否已连接
c_samplerate(di)	self.samplerate	获取采样率（Hz）
c_last_samplenum(di)	self.saved_samplenum	获取上一次 c_wait() 返回的采样号
c_init_pin(di, ch)	self.initial_pin[ch]	获取通道初始引脚值

示例：

```
int has_reset = c_has_ch(di, 1);
uint64_t sr = c_samplerate(di);
uint64_t last_ss = c_last_samplenum(di);
uint8_t init_val = c_init_pin(di, 0);
```

2.7 `c_reg_out()` / `c_reg_meta()` — 输出注册

```
int c_reg_out(struct srd_decoder_inst *di, int output_type, const char *proto_id);
int c_reg_meta(struct srd_decoder_inst *di, int output_type, const char *proto_id,
               const char *meta_type, const char *meta_name, const char *meta_descr);
```

语义：注册解码器输出通道，返回输出 ID（从 0 递增）。后续 `c_put()` / `c_proto()` 等调用使用此 ID。

Python 等价：`self.register(output_type, proto_id)` / `self.register_meta(output_type, ...)`

输出类型：

常量	值	说明
SRD_OUTPUT_ANN	0	注释输出
SRD_OUTPUT_PROTO	1	协议输出（用于 C□C 堆叠）
SRD_OUTPUT_BINARY	2	二进制输出
SRD_OUTPUT_META	3	元数据输出
SRD_OUTPUT_LOGIC	4	逻辑输出

`c_reg_meta` **参数**：

参数	说明
meta_type	"int" 或 "float"/"double"
meta_name	元数据名称
meta_descr	元数据描述

示例：

```
int out_ann    = c_reg_out(di, SRD_OUTPUT_ANN, "i2c");
int out_proto  = c_reg_out(di, SRD_OUTPUT_PROTO, "i2c");
int out_binary = c_reg_out(di, SRD_OUTPUT_BINARY, "i2c");
int out_bitrate = c_reg_meta(di, SRD_OUTPUT_META, "i2c", "int", "bitrate", "Bitrate");
```

2.8 `C_DECODER_STATE` / `C_DECODER_DEFINE` — 状态与定义宏

`C_DECODER_STATE(name, fields)`

```
#define C_DECODER_STATE(name, fields) \
    typedef struct name##_s fields name##_s; \
    static void name##_reset(struct srd_decoder_inst *di) { ... } \
    static void name##_destroy(struct srd_decoder_inst *di) { ... }
```

语义：自动生成状态结构体 `typedef`、`reset` 函数（`calloc` 初始化）和 `destroy` 函数（`free` 释放）。

- 生成类型名：`name##_s`（如 `i2c_s`）
- 生成 `reset`：`name##_reset`（如 `i2c_reset`）
- 生成 `destroy`：`name##_destroy`（如 `i2c_destroy`）

注意：如果状态结构体包含需要特殊释放的资源（如 `GArray *`），应自行实现 `reset`/`destroy` 函数，不使用自动生成的版本。

示例：

```
C_DECODER_STATE(mydec, {
    int state;
    int bitcount;
    uint8_t databyte;
    uint64_t ss_byte;
    int out_ann;
});
//   : mydec_s, mydec_reset(), mydec_destroy()
```

`C_DECODER_DEFINE(dec_name, ...)`

```
#define C_DECODER_DEFINE(dec_name, ...) \
    static struct srd_c_decoder dec_name##_def = { __VA_ARGS__ }; \
    SRD_C_DECODER_EXPORT struct srd_c_decoder *srd_c_decoder_entry(void) { \
        return &dec;_name##_def; \
    } \
    SRD_C_DECODER_EXPORT int srd_c_decoder_api_version(void) { \
        return SRD_C_DECODER_API_VERSION; \
    }
```

语义：自动生成 `srd_c_decoder` 结构体定义和 DLL 入口函数。当前项目中大多数解码器选择手动定义结构体和入口函数，以便在 `srd_c_decoder_entry()` 中设置选项默认值。

2.9 `decode_upper` 回调

```
void (*decode_upper)(struct srd_decoder_inst *di,
                     uint64_t start_sample, uint64_t end_sample,
                     const char *cmd, const c_field *fields, int n_fields);
```

语义：当此解码器堆叠在另一个 C 解码器之上时，下层解码器通过 `c_proto()` 发送的协议数据会触发此回调。`cmd` 为协议命令字符串，`fields` 为 `c_field` 数组，`n_fields` 为字段数量。

Python 等价：Python 解码器中通过 `self.wait()` 隐式接收，无需显式回调。

示例：

```
static void mydec_recv_proto(struct srd_decoder_inst *di,
                             uint64_t start_sample, uint64_t end_sample,
                             const char *cmd, const c_field *fields, int n_fields)
{
    mydec_state *s = (mydec_state *)c_decoder_get_private(di);
    s->ss = start_sample;
    s->es = end_sample;

    if (strcmp(cmd, "ADDRESS WRITE") == 0) {
        uint8_t addr = (n_fields > 0) ? fields[0].u8 : 0;
        //     ...
    } else if (strcmp(cmd, "DATA READ") == 0) {
        uint8_t data = (n_fields > 0) ? fields[0].u8 : 0;
        //     ...
    }
}
```

3. Python→C 翻译规则对照表

3.1 wait 条件映射

Python	C	说明
self.wait({0: 'r'})	c_wait(di, CW_R(0), CW_END)	单通道上升沿
self.wait({0: 'f'})	c_wait(di, CW_F(0), CW_END)	单通道下降沿
self.wait({0: 'h', 1: 'f'})	c_wait(di, CW_H(0), CW_F(1), CW_END)	AND 条件：SCL高且SDA下降
self.wait([{0:'r'}, {0:'h',1:'f'}])	c_wait(di, CW_R(0), CW_OR, CW_H(0), CW_F(1), CW_END)	OR 条件组
self.wait({'skip': 100})	c_wait(di, CW_SKIP(100), CW_END)	跳过采样

Python	C	说明
self.wait()	c_wait(di, CW_END)	无条件前进

3.2 核心属性映射

Python	C	说明
self.samplenum	di_samplenum(di)	当前采样号
self.matched	di_matched(di)	条件匹配（Python 为 tuple，C 为 uint64_t 位掩码）
self.matched[0]	di_matched(di) & (1ULL << 0)	检查第 0 个 OR 组
self.matched[1]	di_matched(di) & (1ULL << 1)	检查第 1 个 OR 组
self.saved_samplenum	c_last_samplenum(di)	上一次 wait 的采样号
self.samplerate	c_samplerate(di)	采样率
self.initial_pin	c_init_pin(di, ch)	初始引脚值
self.has_channel(ch)	c_has_ch(di, ch)	通道是否连接

3.3 输出映射

Python	C	说明
self.put(ss, es, out, [cls, ['Long','Mid','Short']])	c_put(di, ss, es, out, cls, "Long", "Mid", "Short")	多级文本注释
self.put(ss, es, out, [cls, ['Text']])	c_put(di, ss, es, out, cls, "Text")	单级文本注释
self.putp(ss, es, cmd, val)	c_proto(di, ss, es, out, cmd, C_U8(val), C_END)	协议输出

3.4 选项映射

Python	C	说明
self.options['divider']	c_opt_int(di, "divider", 0)	整数选项
self.options['edge']	c_opt_str(di, "edge", "any")	字符串选项
self.options['threshold']	c_opt_dbl(di, "threshold", 1.5)	浮点选项
self.options['invert'] == 'yes'	c_opt_bool(di, "invert", 0)	布尔选项

3.5 通道与引脚映射

Python	C	说明
self.has_channel(1)	c_has_ch(di, 1)	检查可选通道
self.initial_pin[0]	c_init_pin(di, 0)	通道 0 初始值
引脚值（隐式）	c_pin(di, ch)	读取当前引脚电平

3.6 注册输出映射

Python	C	说明
self.register(srd.OUTPUT_ANN)	c_reg_out(di, SRD_OUTPUT_ANN, "proto_id")	注释输出
self.register(srd.OUTPUT_PYTHON)	c_reg_out(di, SRD_OUTPUT_PROTO, "proto_id")	协议输出
self.register(srd.OUTPUT_BINARY)	c_reg_out(di, SRD_OUTPUT_BINARY, "proto_id")	二进制输出
self.register_meta(srd.OUTPUT_META, ...)	c_reg_meta(di, SRD_OUTPUT_META, ...)	元数据输出

4. 常见陷阱与修复方法

陷阱 1：顺序 `c_wait` 导致逻辑错误

错误：将 OR 条件拆分为多个 `c_wait()` 调用。

```
//         c_wait          OR
c_wait(di, CW_R(0), CW_END);   //    SCL
c_wait(di, CW_H(0), CW_F(1), CW_END);  //    SCL +SDA
```

正确：使用 `CW_OR` 合并为一次 `c_wait()`。

```
//      OR
c_wait(di, CW_R(0), CW_OR, CW_H(0), CW_F(1), CW_END);
```

原因：`c_wait()` 是阻塞调用，每次调用都会推进采样指针。两次顺序调用意味着必须先满足第一个条件，再满足第二个条件，而非"任一条件满足"。

陷阱 2：`di_matched()` 是整数位掩码，不是元组

错误：将 `di_matched()` 当作数组或元组访问。

```
//      di_matched    uint64_t
if (di_matched(di)[0]) { ... }
```

正确：使用位运算检查。

```
//              N   OR
if (di_matched(di) & (1ULL << 0)) { ... }  //   0   OR
if (di_matched(di) & (1ULL << 1)) { ... }  //   1   OR
if (di_matched(di) & (1ULL << 2)) { ... }  //   2   OR
```

原因：Python 中 `self.matched` 是 `(True, False, True)` 这样的元组，C 中为了性能使用 `uint64_t` 位掩码，每一位对应一个 OR 组。

陷阱 3：`CW_SKIP(0)` 与 `CW_END` 语义不同

错误：使用 `CW_SKIP(0)` 代替 `CW_END` 终止条件列表。

```
//      CW_SKIP(0)
c_wait(di, CW_R(0), CW_SKIP(0));
```

正确：始终使用 `CW_END` 终止条件列表。

```
//
c_wait(di, CW_R(0), CW_END);
```

原因：`CW_SKIP(0)` 表示跳过 0 个采样（实际上是一个有效的条件项，虽然效果等同于前进一个采样），而 `CW_END`（值为 -2）才是条件列表的终止标记。混淆两者会导致 `c_wait()` 继续读取栈上的垃圾值作为条件。

陷阱 4：`c_proto()` 必须以 `C_END` 终止，不能用 `NULL`

错误：使用 `NULL` 作为 `c_proto()` 的终止标记。

```
//      NULL    c_field
c_proto(di, ss, es, out, "DATA", C_U8(d), NULL);
```

正确：始终使用 `C_END` 宏终止。

```
//
c_proto(di, ss, es, out, "DATA", C_U8(d), C_END);
```

原因：`c_proto()` 内部通过 `va_arg(ap, c_field)` 逐个读取参数，并检查 `f.type == C_FIELD_SENTINEL` 来判断结束。`NULL` 不是 `c_field` 结构体，传入会导致未定义行为或内存错误。

陷阱 5：十六进制格式大小写差异

错误：C 的 `%X` 输出大写十六进制，Python 的 `%x` 输出小写十六进制。

```
//   Python     %x    "0x0a" C   %X    "0x0A"
snprintf(buf, sizeof(buf), "0x%02X", val);
```

修复：如果需要与 Python 保持一致的小写格式，使用 `%x`。

```
//     Python %x
snprintf(buf, sizeof(buf), "0x%02x", val);
```

注意：`C_ANN_PUT_VAL` 宏内部使用 `%X`（大写），如果需要小写，需手动构造注释。

陷阱 6：十六进制位序差异

问题：Python 协议解码器通常以 MSB-first 顺序输出十六进制数据，而 C 代码中直接按字节顺序输出时可能产生 LSB-first 结果。

示例：I²C 地址字节 `0b10100000`： - Python（MSB-first）：地址为 `0x50` - C（如果直接按位拼接后未右移）：地址为 `0x41`（错误）

修复：确保位移方向与协议规范一致。

```
//   I²C    8       1    R/W   7
uint8_t addr = databyte >> 1;  //      R/W
```

陷阱 7：`reduce_bus` 位序差异

问题：Python 解码器中 `self.bits.insert(0, ...)` 实现的是 LSB-first 存储（新元素插入到列表头部），翻译为 C 数组时需要手动移位。

Python 原始代码：

```
self.bits.insert(0, [sda, samplenum, samplenum])
```

C 正确翻译：

```
//                   0
for (int i = 7; i > 0; i--)
    s->bits[i] = s->bits[i - 1];
s->bits[0].sda = sda_val;
s->bits[0].ss = samplenum;
```

```
s->bits[0].es = samplenum;
```

注意：`reversed()` 在 Python 中反转列表，C 中需要反向遍历数组。

陷阱 8：注释类编号必须与 Python 完全一致

问题：C 解码器的注释类编号（`ann_labels` 数组下标）必须与对应 Python 解码器的 `annotations` 元组顺序完全一致，否则前端 UI 会显示错误的注释类型。

错误：

```
//      Python   ACK=3, NACK=4 C
enum { ANN_NACK = 3, ANN_ACK = 4, ... };
```

正确：

```
//       Python annotations
enum { ANN_START = 0, ANN_REPEAT_START = 1, ANN_STOP = 2,
       ANN_ACK = 3, ANN_NACK = 4, ... };
```

陷阱 9：多级文本变体格式差异

Python：注释文本为列表 `['Long text', 'Mid text', 'Short text']`。

C：`c_put()` 的变参直接传递多个字符串。

```
# Python
self.put(ss, es, self.out_ann, [cls, ['Write', 'Wr', 'W']])

// C —              Python
c_put(di, ss, es, out_ann, cls, "Write", "Wr", "W");
```

注意：`c_put()` 内部自动在变参列表末尾追加 `NULL` 终止符，无需手动添加。

陷阱 10：逐采样循环导致超时

错误：在 `decode()` 函数中使用逐采样循环而非 `c_wait()` 条件等待。

```
//
while (1) {
    c_wait(di, CW_END);  //
    if (c_pin(di, 0) == 1 && c_pin(di, 1) == 0) {
        //   ...
        break;
    }
}
```

正确：使用 `c_wait()` 的条件等待机制，让引擎直接跳转到满足条件的采样点。

```
//
while (1) {
    int ret = c_wait(di, CW_H(0), CW_L(1), CW_END);
    if (ret != SRD_OK) return;
    //   ...
}
```

原因：`c_wait()` 的条件等待机制由引擎内部优化，可以快速跳过不满足条件的大量采样。逐采样循环则需要对每个采样执行回调，性能差距可达数千倍。

陷阱 11：忘记检查 `c_wait()` 返回值

错误：不检查 `c_wait()` 返回值。

```
//
c_wait(di, CW_R(0), CW_END);
//     ...
```

正确：始终检查返回值。

```
//
int ret = c_wait(di, CW_R(0), CW_END);
if (ret != SRD_OK)
    return;  //
```

陷阱 12：堆叠解码器的 `decode()` 函数可以为空

说明：当解码器作为堆叠上层（通过 `decode_upper` 接收数据）时，`decode()` 函数不需要实现任何逻辑，因为数据通过 `decode_upper` 回调传入，而非通过 `c_wait()` 从原始采样流读取。

```
//          decode
static void lm75_decode(struct srd_decoder_inst *di)
{
    (void)di;
    //                decode_upper
}
```

5. 完整解码器骨架代码模板

以下模板基于 `counter_c.c` 模式，可直接复制使用：

```
#include
#include
#include
#include
#include "libsigrokdecode.h"

/* ===== 1.       ===== */
enum {
    ANN_XXX = 0,
    ANN_YYY,
    ANN_ZZZ,
    NUM_ANN,
};

/* ===== 2.       ===== */
typedef struct {
    int state;
    uint64_t samplerate;
    int out_ann;
} mydec_state;

/* ===== 3.      ===== */
static struct srd_channel mydec_channels[] = {
    {"clk", "CLK", "Clock line", 0, SRD_CHANNEL_SCLK, "dec_mydec_chan_clk"},
    {"data", "DATA", "Data line", 1, SRD_CHANNEL_SDATA, "dec_mydec_chan_data"},
};

static struct srd_channel mydec_optional_channels[] = {
    {"cs", "CS", "Chip select", 0, SRD_CHANNEL_SDATA, "dec_mydec_opt_chan_cs"},
};

/* ===== 4.      ===== */
static struct srd_decoder_option mydec_options[] = {
```

```
    {"bitorder", "dec_mydec_opt_bitorder", "Bit order", NULL, NULL},
    {"wordsize", "dec_mydec_opt_wordsize", "Word size in bits", NULL, NULL},
};

/* ===== 5.   /  /   ===== */
static const char *mydec_inputs[] = {"logic", NULL};
static const char *mydec_outputs[] = {"mydec", NULL};
static const char *mydec_tags[] = {"Embedded/industrial", NULL};

/* ===== 6.      ===== */
static const char *mydec_ann_labels[][3] = {
    {"", "xxx", "XXX description"},
    {"", "yyy", "YYY description"},
    {"", "zzz", "ZZZ description"},
};

/* ===== 7.     ===== */
static const int mydec_row_xxx_classes[] = {ANN_XXX};
static const int mydec_row_yyy_classes[] = {ANN_YYY};
static const int mydec_row_zzz_classes[] = {ANN_ZZZ};
static const struct srd_c_ann_row mydec_ann_rows[] = {
    {"xxx", "XXX", mydec_row_xxx_classes, 1},
    {"yyy", "YYY", mydec_row_yyy_classes, 1},
    {"zzz", "ZZZ", mydec_row_zzz_classes, 1},
};

/* ===== 8.        ===== */
#define CH_CLK  0
#define CH_DATA 1
#define CH_CS   2

/* ===== 9. reset    ===== */
static void mydec_reset(struct srd_decoder_inst *di)
{
    if (!c_decoder_get_private(di)) {
        c_decoder_set_private(di, g_malloc0(sizeof(mydec_state)));
    }
    mydec_state *s = (mydec_state *)c_decoder_get_private(di);
    memset(s, 0, sizeof(mydec_state));
}

/* ===== 10. start    ===== */
static void mydec_start(struct srd_decoder_inst *di)
{
    mydec_state *s = (mydec_state *)c_decoder_get_private(di);
    s->out_ann = c_reg_out(di, SRD_OUTPUT_ANN, "mydec");
    s->samplerate = c_samplerate(di);

    const char *bitorder = c_opt_str(di, "bitorder", "msb");
    int wordsize = (int)c_opt_int(di, "wordsize", 8);
    (void)bitorder;
    (void)wordsize;
}

/* ===== 11. decode    —       ===== */
static void mydec_decode(struct srd_decoder_inst *di)
{
    mydec_state *s = (mydec_state *)c_decoder_get_private(di);
    int has_cs = c_has_ch(di, CH_CS);
```

```
    while (1) {
        int ret;
        if (has_cs) {
            ret = c_wait(di, CW_R(CH_CLK), CW_OR, CW_F(CH_CS), CW_END);
        } else {
            ret = c_wait(di, CW_R(CH_CLK), CW_END);
        }
        if (ret != SRD_OK)
            return;

        if (has_cs && (di_matched(di) & (1ULL << 1))) {
            // CS
            continue;
        }

        //
        int data_val = c_pin(di, CH_DATA);
        (void)data_val;

        //
        c_put(di, di_samplenum(di), di_samplenum(di), s->out_ann, ANN_XXX, "Text");
    }
}

/* ===== 12. destroy    ===== */
static void mydec_destroy(struct srd_decoder_inst *di)
{
    void *priv = c_decoder_get_private(di);
    if (priv) {
        g_free(priv);
        c_decoder_set_private(di, NULL);
    }
}

/* ===== 13.        ===== */
struct srd_c_decoder mydec_c_decoder = {
    .id = "mydec_c",
    .name = "MyDec(C)",
    .longname = "My Protocol Decoder (C)",
    .desc = "My protocol decoder C implementation",
    .license = "gplv2+",
    .channels = mydec_channels,
    .num_channels = 2,
    .optional_channels = mydec_optional_channels,
    .num_optional_channels = 1,
    .options = mydec_options,
    .num_options = 2,
    .num_annotations = NUM_ANN,
    .ann_labels = mydec_ann_labels,
    .num_annotation_rows = 3,
    .annotation_rows = mydec_ann_rows,
    .inputs = mydec_inputs,
    .num_inputs = 1,
    .outputs = mydec_outputs,
    .num_outputs = 1,
    .binary = NULL,
    .num_binary = 0,
    .tags = mydec_tags,
    .num_tags = 1,
    .reset = mydec_reset,
```

```
    .start = mydec_start,
    .decode = mydec_decode,
    .destroy = mydec_destroy,
    .state_size = 0,
};

/* ===== 14. DLL      ===== */
SRD_C_DECODER_EXPORT struct srd_c_decoder *srd_c_decoder_entry(void)
{
    //
    GSList *bitorder_vals = NULL;
    bitorder_vals = g_slist_append(bitorder_vals, g_variant_new_string("msb"));
    bitorder_vals = g_slist_append(bitorder_vals, g_variant_new_string("lsb"));
    mydec_options[0].def = g_variant_new_string("msb");
    mydec_options[0].values = bitorder_vals;

    //
    GSList *wordsize_vals = NULL;
    wordsize_vals = g_slist_append(wordsize_vals, g_variant_new_int64(8));
    wordsize_vals = g_slist_append(wordsize_vals, g_variant_new_int64(16));
    wordsize_vals = g_slist_append(wordsize_vals, g_variant_new_int64(32));
    mydec_options[1].def = g_variant_new_int64(8);
    mydec_options[1].values = wordsize_vals;

    return &mydec;_c_decoder;
}

SRD_C_DECODER_EXPORT int srd_c_decoder_api_version(void)
{
    return SRD_C_DECODER_API_VERSION;
}
```

构建集成

将新解码器添加到 `CMakeLists.txt` 的 `C_DECODERS` 列表中：

```
set(C_DECODERS
    ...
    mydec_c
    ...
)
```

然后运行 `build_incremental.cmd` 重新构建。

6. 堆叠解码器示例

堆叠解码器（Stacked Decoder）是指一个解码器以上层解码器的协议输出作为输入。例如 `lm75_c` 堆叠在 `i2c_c` 之上，`i2c_c` 输出 I²C 协议数据，`lm75_c` 接收并解析为温度值。

6.1 堆叠架构

```
┌──────────────┐
│   lm75_c     │           decode_upper
│              │  inputs = ["i2c"]
│  decode()    │  ←
│  decode_upper() │ ←    i2c_c   c_proto()
└──────┬───────┘
       │ c_proto()
       ▼
```

```
┌──────────────┐
│   i2c_c      │
│              │  inputs = ["logic"], outputs = ["i2c"]
│  decode()    │  ← c_wait() + c_pin()
│  c_proto()   │  ←
└──────────────┘
```

6.2 下层解码器：发送协议数据

下层解码器（如 `i2c_c`）通过 `c_proto()` 向上层发送协议数据：

```
// i2c_c   start    —
static void i2c_start(struct srd_decoder_inst *di)
{
    i2c_s *s = (i2c_s *)c_decoder_get_private(di);
    s->out_ann    = c_reg_out(di, SRD_OUTPUT_ANN, "i2c");
    s->out_binary = c_reg_out(di, SRD_OUTPUT_BINARY, "i2c");
    s->out_python = c_reg_out(di, SRD_OUTPUT_PROTO, "i2c");  // ←
    s->out_bitrate = c_reg_meta(di, SRD_OUTPUT_META, "i2c", "int", "bitrate", "Bitrate");
    // ...
}

// i2c_c   decode    —
static void i2c_handle_start(struct srd_decoder_inst *di, i2c_s *s)
{
    uint64_t samplenum = di_samplenum(di);

    //
    c_put(di, samplenum, samplenum, s->out_ann, ANN_START, "Start", "S");

    //        —
    c_proto(di, samplenum, samplenum, s->out_python, "START", C_END);
}

//
static void i2c_handle_address(struct srd_decoder_inst *di, i2c_s *s)
{
    // ...
    c_proto(di, s->ss_byte, byte_end, s->out_python,
            s->wr ? "ADDRESS WRITE" : "ADDRESS READ",
            C_U8(d), C_END);
}

//
c_proto(di, s->ss_byte, byte_end, s->out_python,
        s->wr ? "DATA WRITE" : "DATA READ",
        C_U8(d), C_END);

//    ACK/NACK
c_proto(di, samplenum, ack_end, s->out_python, "ACK", C_END);
c_proto(di, samplenum, ack_end, s->out_python, "NACK", C_END);

//    STOP
c_proto(di, samplenum, samplenum, s->out_python, "STOP", C_END);
```

关键点： - `outputs` 数组必须包含协议 ID（如 `"i2c"`） - `c_reg_out(di, SRD_OUTPUT_PROTO, "i2c")` 注册协议输出 - `c_proto()` 的 `cmd` 字符串必须与上层解码器的 `decode_upper` 中 `strcmp(cmd, ...)` 匹配

6.3 上层解码器：接收协议数据

上层解码器（如 `lm75_c`）通过 `decode_upper` 回调接收协议数据：

```
// lm75_c       —        outputs
static const char *lm75_inputs[] = {"i2c", NULL};

// lm75_c
// channels = NULL, num_channels = 0

// lm75_c
enum lm75_state {
    LM75_IDLE,
    LM75_GET_SLAVE_ADDR,
    LM75_READ_REGS,
    LM75_WRITE_REGS,
};

// decode_upper    —    i2c_c
static void lm75_recv_proto(struct srd_decoder_inst *di,
                            uint64_t start_sample, uint64_t end_sample,
                            const char *cmd, const c_field *fields, int n_fields)
{
    lm75_state *s = (lm75_state *)c_decoder_get_private(di);
    if (!s)
        return;

    s->ss = start_sample;
    s->es = end_sample;

    if (s->state == LM75_IDLE) {
        if (strcmp(cmd, "START") != 0)
            return;
        s->state = LM75_GET_SLAVE_ADDR;
    } else if (s->state == LM75_GET_SLAVE_ADDR) {
        if (strcmp(cmd, "ADDRESS READ") == 0 || strcmp(cmd, "ADDRESS WRITE") == 0) {
            //      fields[0].u8
            uint8_t addr = (n_fields > 0) ? fields[0].u8 : 0;
            lm75_warn_upon_invalid_slave(di, s, addr);
            if (strcmp(cmd, "ADDRESS READ") == 0)
                s->state = LM75_READ_REGS;
            else
                s->state = LM75_WRITE_REGS;
            s->num_databytes = 0;
        }
    } else if (s->state == LM75_READ_REGS || s->state == LM75_WRITE_REGS) {
        const char *rw = (s->state == LM75_READ_REGS) ? "READ" : "WRITE";

        if (strcmp(cmd, "DATA READ") == 0 || strcmp(cmd, "DATA WRITE") == 0) {
            uint8_t databyte = (n_fields > 0) ? fields[0].u8 : 0;
            //       ...
        } else if (strcmp(cmd, "STOP") == 0) {
            s->state = LM75_IDLE;
        }
    }
}

// decode        —      decode_upper
static void lm75_decode(struct srd_decoder_inst *di)
{
    (void)di;
}
```

```
//        –
struct srd_c_decoder lm75_c_decoder = {
    // ...
    .channels = NULL,           //
    .num_channels = 0,
    .optional_channels = NULL,
    .num_optional_channels = 0,
    .inputs = lm75_inputs,      //     "i2c"
    .num_inputs = 1,
    .outputs = NULL,            //
    .num_outputs = 0,
    .decode = lm75_decode,      //
    .decode_upper = lm75_recv_proto,  // ←
    // ...
};
```

6.4 `c_field` 数组解析

在 `decode_upper` 回调中，`fields` 参数是 `c_field` 结构体数组，`n_fields` 为字段数量。根据 `type` 字段判断数据类型并访问对应联合体成员：

```
static void my_recv_proto(struct srd_decoder_inst *di,
                          uint64_t start_sample, uint64_t end_sample,
                          const char *cmd, const c_field *fields, int n_fields)
{
    for (int i = 0; i < n_fields; i++) {
        switch (fields[i].type) {
        case C_FIELD_U8:   printf("U8: %u\n", fields[i].u8);   break;
        case C_FIELD_U16:  printf("U16: %u\n", fields[i].u16); break;
        case C_FIELD_U32:  printf("U32: %u\n", fields[i].u32); break;
        case C_FIELD_U64:  printf("U64: %llu\n", (unsigned long long)fields[i].u64); break;
        case C_FIELD_I8:   printf("I8: %d\n", fields[i].i8);   break;
        case C_FIELD_I16:  printf("I16: %d\n", fields[i].i16); break;
        case C_FIELD_I32:  printf("I32: %d\n", fields[i].i32); break;
        case C_FIELD_I64:  printf("I64: %lld\n", (long long)fields[i].i64); break;
        case C_FIELD_F64:  printf("F64: %f\n", fields[i].f64); break;
        case C_FIELD_STR:  printf("STR: %s\n", fields[i].str); break;
        case C_FIELD_BYTES:
            printf("BYTES: len=%u\n", fields[i].bytes.len);
            break;
        }
    }
}
```

6.5 堆叠解码器注册清单

字段	下层解码器 (i2c_c)	上层解码器 (lm75_c)
channels	SCL, SDA	NULL （无通道）
inputs	{"logic", NULL}	{"i2c", NULL}
outputs	{"i2c", NULL}	NULL （无输出）
decode()	从采样流解码	可以为空
decode_upper	NULL	接收协议数据的回调
c_reg_out(PROTO)	注册协议输出	不需要

字段	下层解码器 (i2c_c)	上层解码器 (lm75_c)
c_proto()	发送协议数据	不调用
c_put()	输出注释	输出注释

6.6 三层堆叠示例

更复杂的堆叠可以有三层或更多。例如 `i2c_c` → `eeprom24xx_c` → 应用层解码器：

```
┌──────────────────┐
│            │  decode_upper    eeprom24xx_c
└──────────────────┘
        │ c_proto("DATA", C_U8(byte), C_END)
        ▼
┌──────────────────┐
│  eeprom24xx_c    │  decode_upper    i2c_c
│  inputs=["i2c"]  │  outputs=["eeprom24xx"]
│  decode_upper()  │  c_proto()
└──────────────────┘
        │ c_proto("ADDRESS WRITE", C_U8(addr), C_END)
        ▼
┌──────────────────┐
│  i2c_c           │
│  inputs=["logic"]│  outputs=["i2c"]
│  decode()        │  c_proto()
└──────────────────┘
```

中间层解码器（如 `eeprom24xx_c`）同时具有 `decode_upper`（接收下层数据）和 `c_proto()`（向上层发送数据）。

附录 A：`srd_c_decoder` 结构体字段完整参考

```
struct srd_c_decoder {
    const char *id;              //     ID        "spi_c"
    const char *name;            //       "SPI(C)"
    const char *longname;        //       "Serial Peripheral Interface"
    const char *desc;            //
    const char *license;         //     "gplv2+"   "gplv3+"

    const struct srd_channel *channels;          //
    int num_channels;                            //
    const struct srd_channel *optional_channels;  //
    int num_optional_channels;                   //
    const struct srd_decoder_option *options;    //
    int num_options;                             //

    int num_annotations;                         //
    const char *(*ann_labels)[3];               //      [ ,  ,  ]
    int num_annotation_rows;                     //
    const struct srd_c_ann_row *annotation_rows; //

    const char **inputs;        //      ID
    int num_inputs;             //
    const char **outputs;       //      ID
    int num_outputs;            //
    const struct srd_decoder_binary *binary;  //
    int num_binary;                          //
    const char **tags;          //
```

```
    int num_tags;               //

    size_t state_size;          //         C_DECODER_STATE

    void (*reset)(struct srd_decoder_inst *di);     //
    void (*start)(struct srd_decoder_inst *di);     //
    void (*decode)(struct srd_decoder_inst *di);    //
    void (*end)(struct srd_decoder_inst *di);       //         NULL
    void (*metadata)(struct srd_decoder_inst *di, int key, uint64_t value);  //          NULL
    void (*destroy)(struct srd_decoder_inst *di);   //
    void (*decode_upper)(struct srd_decoder_inst *di,  //           NULL
                         uint64_t start_sample, uint64_t end_sample,
                         const char *cmd, const c_field *fields, int n_fields);
};
```

附录 B：srd_channel 结构体字段

```
struct srd_channel {
    char *id;     //    ID   "scl"
    char *name;   //        "SCL"
    char *desc;   //        "Serial clock line"
    int order;    //        0
    int type;     //      SRD_CHANNEL_SCLK / SRD_CHANNEL_SDATA / SRD_CHANNEL_COMMON
    char *idn;    //        ID    NULL
};
```

附录 C：srd_c_ann_row 结构体字段

```
struct srd_c_ann_row {
    const char *id;           //   ID   "bits"
    const char *desc;         //       "Bits"
    const int *ann_classes;   //
    int num_ann_classes;      //
};
```

附录 D：srd_decoder_option 结构体字段

```
struct srd_decoder_option {
    char *id;        //    ID   "bitorder"
    char *idn;       //        ID
    char *desc;      //
    GVariant *def;   //       srd_c_decoder_entry
    GSList *values;  //         srd_c_decoder_entry
};
```

附录 E：srd_decoder_binary 结构体字段

```
struct srd_decoder_binary {
    int bin_class;         //
    const char *id;        //       ID
    const char *desc;      //
};
```