RustPLC：工业控制数字孪生平台

从形式化验证到虚实混合闭环

RustPLC 项目技术文档 v3

2026 年 2 月

Contents

Part I
验证域 (BC1)：从 DSL 到形式化证明

1 引言：为什么需要 RustPLC

传统 PLC（可编程逻辑控制器）编程使用梯形图、结构化文本（ST）或功能块图（FBD），程序员手动编写控制逻辑，然后通过测试来验证正确性。这种“先写后测”的模式存在根本性缺陷：

- 测试只能发现 **bug**，不能证明没有 **bug**——测试覆盖的是有限路径，而工业系统的状态空间是指数级的。
- 安全约束散落在代码各处——互锁逻辑与业务逻辑混杂，难以审计。
- 时序问题难以复现——传感器延迟、气缸行程时间等物理参数在仿真中容易被忽略。

RustPLC 的核心主张是：

> 不要编写控制程序——声明物理事实和安全意图，让编译器在代码运行之前证明它是安全的。

但 RustPLC 不止于验证。它是一个完整的数字孪生平台，覆盖从源码到物理世界的完整闭环：

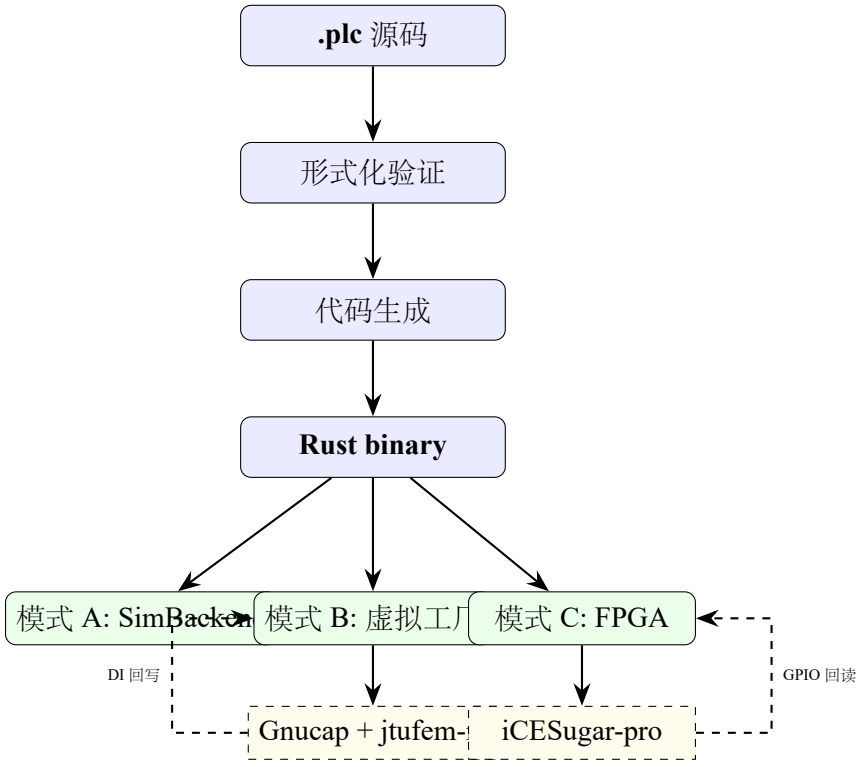


Figure 1: RustPLC 完整闭环：从源码到物理世界

2 第一性原理

RustPLC 的设计建立在以下不可再分的基本原理之上：

2.1 原理一：物理系统是有限状态的

工业控制中的每个设备都有有限的状态集合。气缸只有 “伸出” 和 “缩回” 两个状态，电磁阀只有 “开” 和 “关”，传感器只有 “检测到” 和 “未检测到”。整个系统的状态空间是所有设备状态的笛卡尔积：

$$\mathcal{S} = S_{\text{dev}_1} \times S_{\text{dev}_2} \times \cdots \times S_{\text{dev}_n} \times S_{\text{ctrl}}$$

其中 S_{ctrl} 是控制状态机的状态集。由于每个因子都是有限的，$\mathcal{S}$ 也是有限的——这意味着我们可以用穷举或归纳的方式证明性质。

2.2 原理二：安全性是状态空间上的不变量

“A 缸和 B 缸不能同时伸出” 这样的安全约束，本质上是对状态空间的一个划分：

$$\mathcal{S}_{\text{safe}} = \{s \in \mathcal{S} \mid \neg(\text{cyl\_A.extended} \wedge \text{cyl\_B.extended})\}$$

安全验证的任务就是证明：从初始状态出发，沿任意合法转移路径，系统永远停留在 $\mathcal{S}_{\text{safe}}$ 中。这是一个经典的模型检验（Model Checking）问题。

2.3 原理三：因果律决定信号传播路径

物理信号不能凭空出现。一个传感器要检测到气缸伸出，必须存在完整的因果链：

$$\text{数字输出} \xrightarrow{\text{电气}} \text{电磁阀} \xrightarrow{\text{气动}} \text{气缸} \xrightarrow{\text{物理}} \text{传感器}$$

如果拓扑图中这条路径断开，程序逻辑再正确也无法工作。

2.4 原理四：活性保证系统不会 “卡死”

一个安全但永远不动的系统没有价值。活性（Liveness）要求每个非终态都有出路：要么有超时跳转，要么有明确的 “允许无限等待” 标记（如等待人工按钮）。

2.5 原理五：时序约束来自物理定律

电磁阀有响应时间，气缸有行程时间，电机有加速斜坡。这些不是软件参数，而是物理常数。编译器必须沿拓扑连接链累加这些时间，计算最坏情况关键路径，并与声明的时序约束比较。

3 编译器架构

RustPLC 采用经典的多阶段编译器设计，如图 2 所示。

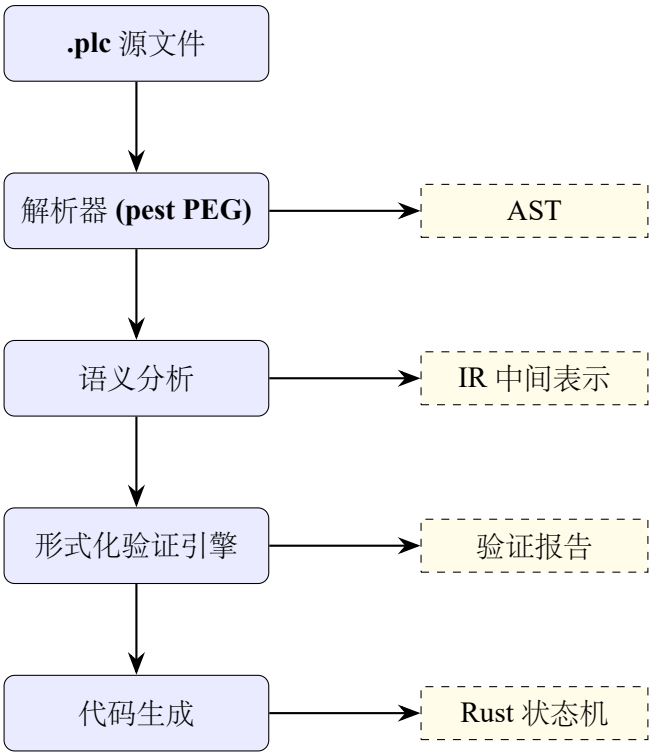


Figure 2: RustPLC 编译流水线（含代码生成阶段）

3.1 阶段一：解析（**Parser**）

使用 `pest` PEG 解析器，将 `.plc` 源文件转换为抽象语法树（AST）。语法文件 `plc.pest` 定义了 150 条规则，覆盖三大段落：

- `[topology]`——设备声明与物理连接
- `[constraints]`——安全/时序/因果约束
- `[tasks]`——控制任务与步骤（状态机）

3.2 阶段二：语义分析（**Semantic Analysis**）

将 AST 降低为四种中间表示：

IR 结构	数据结构	用途
拓扑图	`petgraph::DiGraph`	设备连接关系
状态机	状态 + 转移 + 守卫	控制流建模
约束集	安全/时序/因果规则	验证目标
时序模型	动作 → 时间区间映射	时序分析

Table 1: 四种中间表示

3.3 阶段三：形式化验证

四个独立的验证引擎并行运行：

1. 安全检查器——有界模型检验（BMC）+ k-归纳，可选 Z3 SMT 求解器
2. 活性检查器——强连通分量（SCC）分析 + 死锁检测
3. 时序检查器——最坏情况关键路径分析
4. 因果检查器——拓扑图 BFS 可达性验证

3.4 阶段四：代码生成（**v3** 新增）

验证通过后，编译器将 IR 转换为可执行的 Rust 代码。详见第 8 章。

4 DSL 语言设计

每个 `.plc` 文件由三个段落组成。下面以“传送带冲压系统”为例逐段讲解。

4.1 拓扑段 `[topology]`

声明物理设备及其连接关系：

```
[topology]

device Y0: digital_output
device Y1: digital_output
device X0: digital_input
device X1: digital_input
device X2: digital_input
device X3: digital_input

device start_button: digital_input {
    connected_to: X3
    debounce: 20ms
}

device conveyor_motor: motor {
    connected_to: Y0
    rated_speed: 30rpm
    ramp_time: 100ms
}

device stamp_valve: solenoid_valve {
    connected_to: Y1
    response_time: 15ms
}

device stamp_head: cylinder {
    connected_to: stamp_valve
    stroke_time: 250ms
    retract_time: 200ms
}

```

```
device sensor_in_position: sensor {
    type: proximity
    connected_to: X0
    detects: conveyor_motor.position_A
}

device sensor_stamp_down: sensor {
    type: magnetic
    connected_to: X1
    detects: stamp_head.extended
}

device sensor_stamp_up: sensor {
    type: magnetic
    connected_to: X2
    detects: stamp_head.retracted
}
```

`connected_to` 表示上游连接：`stamp_head` 连接到 `stamp_valve`，意味着电磁阀驱动气缸。编译器据此构建有向拓扑图。

4.2 约束段 [constraints]

声明安全、时序和因果约束：

```
[constraints]

safety: stamp_head.extended conflicts_with conveyor_motor.on
    reason: "冲压头下压时传送带不能运行"

timing: task.cycle must_complete_within 3000ms
    reason: "单个冲压周期不应超过3秒"

causality: Y1 -> stamp_valve -> stamp_head -> sensor_stamp_down
causality: Y1 -> stamp_valve -> stamp_head -> sensor_stamp_up
```

- **safety**：`conflicts_with` 表示两个状态互斥，`requires` 表示蕴含。
- **timing**：`must_complete_within` 设定上界，`must_start_after` 设定下界。
- **causality**：用 `->` 链声明信号传播路径。

4.3 任务段 [tasks]

定义控制逻辑为状态机：

```
[tasks]

task cycle:
    step feed:
        action: set conveyor_motor on
        wait: sensor_in_position == true
```

```
        timeout: 1500ms -> goto fault_handler
    step stop_belt:
        action: set conveyor_motor off
    step press_down:
        action: extend stamp_head
        wait: sensor_stamp_down == true
        timeout: 500ms -> goto fault_handler
    step press_up:
        action: retract stamp_head
        wait: sensor_stamp_up == true
        timeout: 500ms -> goto fault_handler
    on_complete: goto ready

task fault_handler:
    step emergency:
        action: retract stamp_head
        action: set conveyor_motor off
    step report:
        action: log "冲压系统故障：动作超时"
    on_complete: goto ready

task ready:
    step wait_start:
        wait: start_button == true
        allow_indefinite_wait: true
    on_complete: goto cycle
```

5 验证引擎详解

5.1 安全检查器：有界模型检验

安全检查器的核心算法是有界模型检验（Bounded Model Checking, BMC），辅以 k-归纳（k-induction）来获得完备证明。

5.1.1 状态空间建模

系统的具体状态是一个元组：

$$\sigma = (q, d_1, d_2, \ldots, d_n)$$

其中 $q \in Q$ 是控制状态机的当前状态，$d_i \in D_i$ 是第 i 个设备的当前状态。

5.1.2 BMC 搜索

从初始状态 σ_0 出发，BFS 展开所有可达状态，直到深度 k：

$$\text{Reach}_k = \{\sigma \mid \exists \sigma_0 \xrightarrow{t_1} \sigma_1 \xrightarrow{t_2} \cdots \xrightarrow{t_k} \sigma,\ \sigma_0 \in \mathcal{I}\}$$

对每个 `conflicts_with` 约束 ϕ，检查：

$$\forall \sigma \in \text{Reach}_k : \sigma \models \neg\phi$$

5.1.3 k-归纳

如果 BFS 在深度 k 时状态空间已收敛（无新状态），则获得完备证明：

$$\text{Reach}_k = \text{Reach}_{k+1} \implies \text{Reach}_k = \text{Reach}_\infty$$

在我们的传送带冲压示例中，安全检查器在深度 8 获得完备证明，证明了 “冲压头下压时传送带不会运行” 这一安全性质。

5.2 活性检查器：死锁检测

活性检查器执行四项检查：

1. 无超时等待：每个 `wait` 必须有 `timeout` 或 `allow_indefinite_wait`。
2. 不可达声明验证：标记为 `unreachable` 的 `on_complete` 必须所有路径都跳走。
3. 零出度检测：非终态必须有出边。
4. **SCC** 分析：使用 Kosaraju 算法检测可能困住执行的环路。

5.3 时序检查器：关键路径分析

时序检查器沿拓扑连接链累加设备物理参数，计算最坏情况执行时间。

以冲压周期为例：

步骤	关键动作	最坏时间
feed	电机启动 + 等待到位	1500ms（超时上界）
stop_belt	电机停止	100ms
press_down	电磁阀响应 + 气缸行程	500ms（超时上界）
press_up	气缸回程	500ms（超时上界）
总计		**2600ms**

Table 2: 冲压周期最坏情况时序分析

2600ms $<$ 3000ms，时序约束满足。

5.4 因果检查器：信号可达性

因果检查器在拓扑图上执行 BFS，验证声明的因果链中每一跳都有对应的物理连接：

$$\text{Y1} \xrightarrow{\text{electrical}} \text{stamp\_valve} \xrightarrow{\text{pneumatic}} \text{stamp\_head} \xrightarrow{\text{logical}} \text{sensor\_stamp\_down}$$

如果任何一跳断开（例如 `connected_to` 配置错误），编译器会报告断裂位置并给出修复建议。

6 完整示例：传送带冲压系统

6.1 场景描述

一条自动化产线上，工件由传送带送入冲压工位。传感器检测到工件到位后，传送带停止，冲压头下压完成冲压，然后回位，传送带继续运行。

关键安全约束：冲压头下压时传送带绝对不能运行，否则工件移位会导致冲压偏移，损坏模具甚至造成安全事故。

6.2 系统拓扑

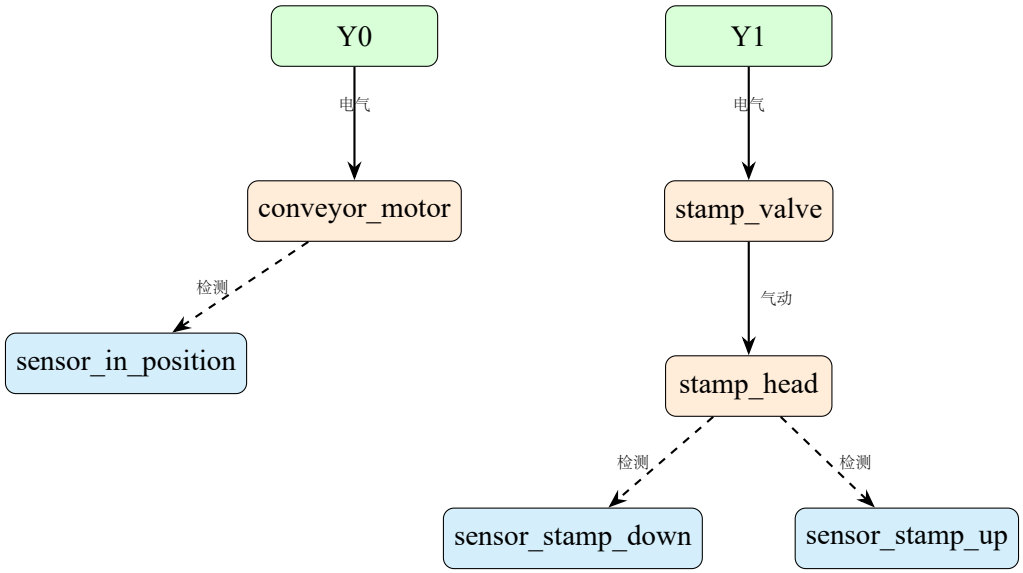


Figure 3: 传送带冲压系统拓扑图

6.3 状态机

6.4 运行验证

在项目根目录执行：

```
$ cargo run -- examples/industrial/conveyor_stamp.plc
```

输出：

```
验证通过：
  - Safety: 完备证明（深度 8）
  - Liveness: 通过
  - Timing: 通过
  - Causality: 通过
```

四项验证全部通过，含义如下：

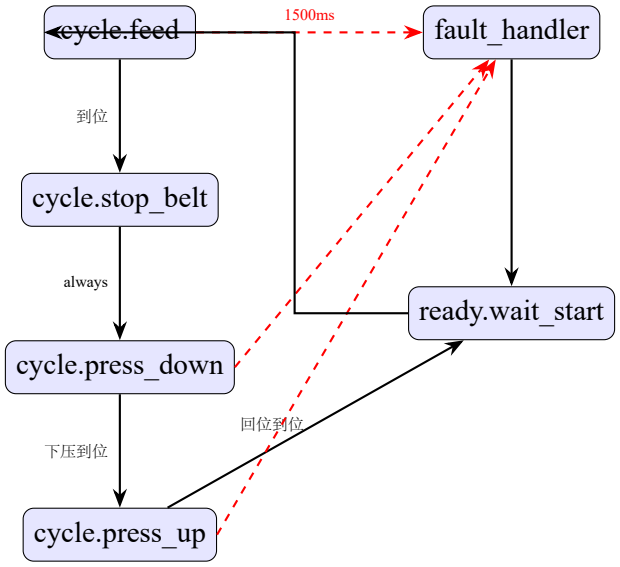


Figure 4: 传送带冲压系统状态机（红色虚线为超时跳转）

- **Safety** 完备证明：在深度 8 的状态空间搜索中，状态已收敛。数学上证明了在任意执行路径上，冲压头下压时传送带都不会运行。
- **Liveness** 通过：所有等待都有超时保护或明确的无限等待许可，不存在死锁。
- **Timing** 通过：最坏情况关键路径 2600ms < 约束上界 3000ms。
- **Causality** 通过：从 Y1 到 sensor_stamp_down 和 sensor_stamp_up 的信号传播路径在拓扑图中完整连通。

7 深度追踪：一条安全约束如何变成数学证明

前面我们从宏观上介绍了编译器架构和验证引擎。但“自动化验证”到底是怎么实现的？本节以传送带冲压系统中的一条安全约束为例，逐函数、逐数据结构地追踪它从 DSL 文本变成数学证明的完整路径。

我们追踪的目标是这一行：

```
safety: stamp_head.extended conflicts_with conveyor_motor.on
```

整个过程经历 5 个阶段，涉及 12 个关键函数和 8 个数据结构。

7.1 阶段一：文本 → AST（解析）

`pest` 解析器用 `plc.pest` 中的语法规则匹配文本。`parse_safety_constraint()` 遍历语法树，产出 AST 节点：

```
SafetyConstraint {
    line: 47,
    left:  StateReference { device: "stamp_head",     state: "extended" },
    relation: ConflictsWith,
    right: StateReference { device: "conveyor_motor", state: "on" },
```

```
    reason: Some(" 冲压头下压时传送带不能运行...")
}
```

7.2 阶段二：AST → IR（语义分析）

`build_constraint_set_from_ast()` 对每条 safety 约束执行两步：验证引用合法性（设备是否存在、状态是否合法），然后降低为 IR：

```
SafetyRule {
    left:     StateExpr { device: "stamp_head",     state: "extended" },
    relation: ConflictsWith,
    right:    StateExpr { device: "conveyor_motor", state: "on" },
}
```

7.3 阶段三：IR → SafetyModel（模型构建）

`SafetyModel::from_inputs()` 构建验证模型：

1. `collect_device_domains()` 为每个设备建立有限状态域和数值索引
2. `transition_effects()` 将状态机转移映射为设备状态变更
3. `scc_minimum_depth()` 用 Kosaraju 算法确定搜索深度（本例为 8）

7.4 阶段四：BFS 穷举搜索

`analyze_rule()` 从初始状态 BFS 展开，对每个可达状态检查冲突：

深度	控制状态	**stamp_head**	**conveyor_motor**	冲突?
0	cycle.feed	retracted	off	No
1	cycle.stop_belt	retracted	on	No
2	cycle.press_down	retracted	off	No
3	cycle.press_up	extended	off	No
4	ready.wait_start	retracted	off	No
⋮	⋮	⋮	⋮	⋮

Table 3: BFS 搜索过程（关键设备状态追踪）

深度 3：`stamp_head` 变为 extended 时，`conveyor_motor` 已经是 off。BFS 在深度 8 穷尽所有可达状态，未发现冲突。

7.5 阶段五：证明判定

$\text{Reach}_8 = \text{Reach}_9$，状态空间收敛，获得完备证明：

$$\forall \sigma \in \text{Reach}_\infty : \neg(\text{stamp\_head.extended} \wedge \text{conveyor\_motor.on})$$

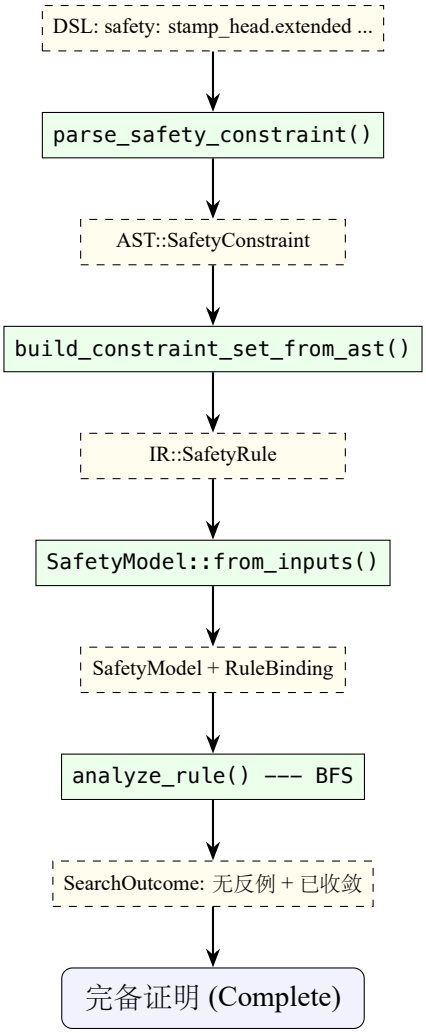


Figure 5: 安全约束从文本到证明的完整数据流

Part II
执行域 (BC2)：从 IR 到确定性执行

8 代码生成：IR → Rust 状态机

验证通过后，编译器将 IR 转换为可独立编译运行的 Rust 项目。代码生成器的设计原则是：生成的代码是纯粹的状态机，没有 **I/O** 副作用。所有 I/O 通过 `HalBackend` trait 完成，使得同一份生成代码可以驱动仿真后端、Modbus 后端或 FPGA 后端。

8.1 生成产物

代码生成器输出一个完整的 Cargo workspace：

```
generated/
├── Cargo.toml           # workspace 根
├── src/
│   ├── main.rs          # 入口：创建 HAL + 启动 ScanCycleEngine
│   ├── plc_state.rs     # PlcState enum + scan_cycle 函数
│   └── device_map.rs    # 设备名 → HAL 地址映射
└── config/
    └── hal.toml         # HAL 后端配置
```

8.2 PlcState enum

每个控制状态机的状态对应一个 enum variant：

```rust
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum PlcState {
    CycleFeed,
    CycleStopBelt,
    CyclePressDown,
    CyclePressUp,
    FaultHandlerEmergency,
    FaultHandlerReport,
    ReadyWaitStart,
}
```

命名规则：`task_name` + `step_name`，驼峰化。编译器保证 enum 的 variant 集合与 IR 中的状态集合一一对应。

8.3 scan_cycle 函数

核心生成产物——一个纯粹的 `match` 状态机：

```rust
pub fn scan_cycle(
    state: &mut PlcState,
    hal: &mut impl HalBackend,
    timers: &mut TimerBank,
) {
```

```
    match state {
        PlcState::CycleFeed => {
            hal.write_digital_output("conveyor_motor", true);
            timers.start(0, 1500); // timer 0: feed timeout
            if hal.read_digital_input("sensor_in_position") {
                timers.stop(0);
                *state = PlcState::CycleStopBelt;
            } else if timers.expired(0) {
                *state = PlcState::FaultHandlerEmergency;
            }
        }
        PlcState::CycleStopBelt => {
            hal.write_digital_output("conveyor_motor", false);
            *state = PlcState::CyclePressDown;
        }
        PlcState::CyclePressDown => {
            hal.write_digital_output("stamp_valve", true);
            timers.start(1, 500);
            if hal.read_digital_input("sensor_stamp_down") {
                timers.stop(1);
                *state = PlcState::CyclePressUp;
            } else if timers.expired(1) {
                *state = PlcState::FaultHandlerEmergency;
            }
        }
        // ... 其余状态类似
    }
}
```

关键设计决策：

- 无 **I/O** 副作用：`scan_cycle` 只通过 `hal` 参数读写 I/O，不直接访问硬件。这使得单元测试可以用 `SimBackend` 驱动。
- 定时器外置：`TimerBank` 由引擎管理，`scan_cycle` 只负责启动/停止/查询。这避免了生成代码中出现时间相关的副作用。
- 单次扫描语义：每次调用 `scan_cycle` 只执行一次状态转移判断，不会在内部循环。这与传统 PLC 的扫描周期语义一致。

9 扫描周期引擎

`ScanCycleEngine` 是 BC2 的核心组件，负责以固定周期驱动状态机：

```
pub struct ScanCycleEngine<H: HalBackend> {
    state: PlcState,
    hal: H,
    timers: TimerBank,
    cycle_ms: u64,
}

impl<H: HalBackend> ScanCycleEngine<H> {
```

```
    pub fn run_cycle(&mut self) -> Result<(), HalError> {
        // 1. 读取所有输入
        self.hal.refresh_inputs()?;

        // 2. 执行状态机
        scan_cycle(&mut self.state, &mut self.hal, &mut
            self.timers);

        // 3. 写出所有输出
        self.hal.flush_outputs()?;

        // 4. 推进定时器
        self.timers.tick(self.cycle_ms);

        Ok(())
    }
}
```

四步扫描周期的顺序是严格的：

1. `refresh_inputs()`——批量读取所有输入信号，建立本周期的一致性快照。
2. `scan_cycle()`——基于快照执行状态转移，产生输出指令。
3. `flush_outputs()`——批量写出所有输出信号。
4. `timers.tick()`——推进所有活跃定时器。

这个顺序保证了：在一个扫描周期内，状态机看到的输入是一致的（不会在执行过程中变化），输出也是原子性的（不会出现半写状态）。这与 IEC 61131-3 标准的扫描周期语义一致。

10 定时器组

`TimerBank` 管理 32 个独立的倒计时定时器：

```
pub struct TimerBank {
    slots: [TimerSlot; 32],
}

pub struct TimerSlot {
    active: bool,
    remaining_ms: u64,
}

impl TimerBank {
    pub fn start(&mut self, id: usize, duration_ms: u64);
    pub fn stop(&mut self, id: usize);
    pub fn expired(&self, id: usize) -> bool;
    pub fn tick(&mut self, elapsed_ms: u64);
}
```

代码生成器为每个 `timeout` 分配一个定时器槽位。`tick()` 在每个扫描周期末尾调用，将所有活跃定时器的剩余时间减去周期时长。

Part III
硬件适配域 (BC3)：HalBackend 与三种模式

11 HalBackend trait 设计

整个平台的核心解耦点是 `HalBackend` trait。它定义了运行时与物理世界之间的契约：

```rust
pub trait HalBackend {
    /// 读取数字输入信号
    fn read_digital_input(&self, device: &str) -> bool;

    /// 写入数字输出信号
    fn write_digital_output(&mut self, device: &str, value:
       bool);

    /// 批量刷新所有输入（从物理/仿真世界读取）
    fn refresh_inputs(&mut self) -> Result<(), HalError>;

    /// 批量写出所有输出（到物理/仿真世界）
    fn flush_outputs(&mut self) -> Result<(), HalError>;
}
```

这个 trait 的设计遵循依赖倒置原则（Dependency Inversion Principle）：高层模块（`ScanCycleEngine`）不依赖低层模块（Modbus/FPGA/内存），两者都依赖抽象（`HalBackend`）。

11.1 为什么是 `&str` 而不是地址？

`read_digital_input` 的参数是设备名（`&str`），而不是 Modbus 地址或 GPIO 编号。这是有意为之的设计：

- 生成的 `scan_cycle` 代码使用设备名，与 DSL 中的名称一致，可读性好。
- 设备名到物理地址的映射由 `DeviceMapping` 配置文件完成，不硬编码在生成代码中。
- 同一份生成代码可以通过不同的配置文件驱动不同的 HAL 后端。

11.2 DeviceMapping 配置

设备名到物理地址的映射通过 TOML 配置文件完成：

```
# config/hal_modbus.toml
[backend]
```

```
type = "modbus_tcp"
host = "192.168.1.100"
port = 502
unit_id = 1

[mapping]
conveyor_motor = { type = "coil", address = 0 }
stamp_valve    = { type = "coil", address = 1 }
sensor_in_position = { type = "discrete_input", address = 0 }
sensor_stamp_down  = { type = "discrete_input", address = 1 }
sensor_stamp_up    = { type = "discrete_input", address = 2 }
start_button       = { type = "discrete_input", address = 3 }
```

12 模式 A：SimBackend

`SimBackend` 是最简单的 HAL 实现，用于 CI 测试和单元测试：

```rust
pub struct SimBackend {
    inputs: HashMap<String, bool>,
    outputs: HashMap<String, bool>,
}

impl HalBackend for SimBackend {
    fn read_digital_input(&self, device: &str) -> bool {
        *self.inputs.get(device).unwrap_or(&false)
    }

    fn write_digital_output(&mut self, device: &str, value:
       bool) {
        self.outputs.insert(device.to_string(), value);
    }

    fn refresh_inputs(&mut self) -> Result<(), HalError> {
        Ok(()) // 内存后端无需刷新
    }

    fn flush_outputs(&mut self) -> Result<(), HalError> {
        Ok(()) // 内存后端无需刷新
    }
}
```

测试代码可以直接操作 `inputs` HashMap 来模拟传感器信号：

```rust
#[test]
fn test_conveyor_stamp_cycle() {
    let mut engine = ScanCycleEngine::new(
        PlcState::ReadyWaitStart,
        SimBackend::new(),
        50, // 50ms cycle
    );

```

```
    // 模拟按下启动按钮
    engine.hal.inputs.insert("start_button".into(), true);
    engine.run_cycle().unwrap();
    assert_eq!(engine.state, PlcState::CycleFeed);

    // 模拟工件到位
    engine.hal.inputs.insert("sensor_in_position".into(), true);
    engine.run_cycle().unwrap();
    assert_eq!(engine.state, PlcState::CycleStopBelt);
}
```

13 模式 B：ModbusBackend（计划中）

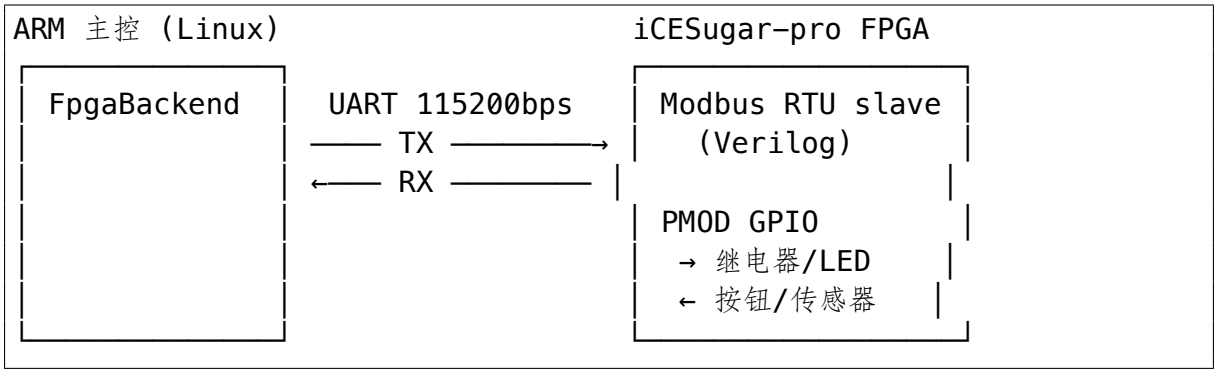
`ModbusBackend` 使用 `tokio-modbus` 库通过 Modbus RTU/TCP 协议与远程 I/O 从站通信：

```
pub struct ModbusBackend {
    ctx: tokio_modbus::client::Context,
    mapping: DeviceMapping,
    input_cache: HashMap<String, bool>,
    output_buffer: HashMap<String, bool>,
}

impl HalBackend for ModbusBackend {
    fn refresh_inputs(&mut self) -> Result<(), HalError> {
        // 批量读取所有 discrete_input 寄存器
        let addrs = self.mapping.discrete_input_range();
        let values = self.ctx.read_discrete_inputs(addrs)?;
        for (name, addr) in self.mapping.inputs() {
            self.input_cache.insert(name, values[addr]);
        }
        Ok(())
    }

    fn flush_outputs(&mut self) -> Result<(), HalError> {
        // 批量写入所有 coil 寄存器
        let coils: Vec<bool> = self.mapping.coil_range()
            .map(|addr| self.output_buffer.get(&addr).copied()
                 .unwrap_or(false))
            .collect();
        self.ctx.write_multiple_coils(0, &coils)?;
        Ok(())
    }
    // ...
}
```

关键设计：`refresh_inputs()` 和 `flush_outputs()` 各只发一次 Modbus 请求，而不是每个设备一次。这将 Modbus 通信开销从 $O(n)$ 降到 $O(1)$，对于 50ms 扫描周期至关重要。

14 模式 C：FpgaBackend（计划中）

`FpgaBackend` 通过 UART 与 iCESugar-pro FPGA 通信，FPGA 内部运行硬件 Modbus RTU slave 状态机：



FPGA 的优势是确定性时序：GPIO 翻转在纳秒级完成，不受操作系统调度影响。这对于需要精确时序的安全关键应用至关重要。

Part IV
DDD 架构：限界上下文与集成模式

15 为什么用 DDD

RustPLC 不是一个单体应用，而是一个跨越编译器、运行时、通信协议、电路仿真、力学仿真和 FPGA 硬件的异构系统。传统的分层架构（Layer Architecture）无法表达这些子系统之间的复杂关系——它们不是简单的上下层调用，而是通过不同的协议和数据格式进行双向通信。

Domain-Driven Design（DDD）的限界上下文（Bounded Context）提供了更好的组织方式：每个上下文有明确的职责边界、领域语言和集成契约。

16 五个限界上下文

17 集成模式

上下文之间的集成不是随意的——每种集成关系都对应 DDD 中的一种标准模式：

18 Crate 映射

每个限界上下文对应一个或多个 Rust crate：

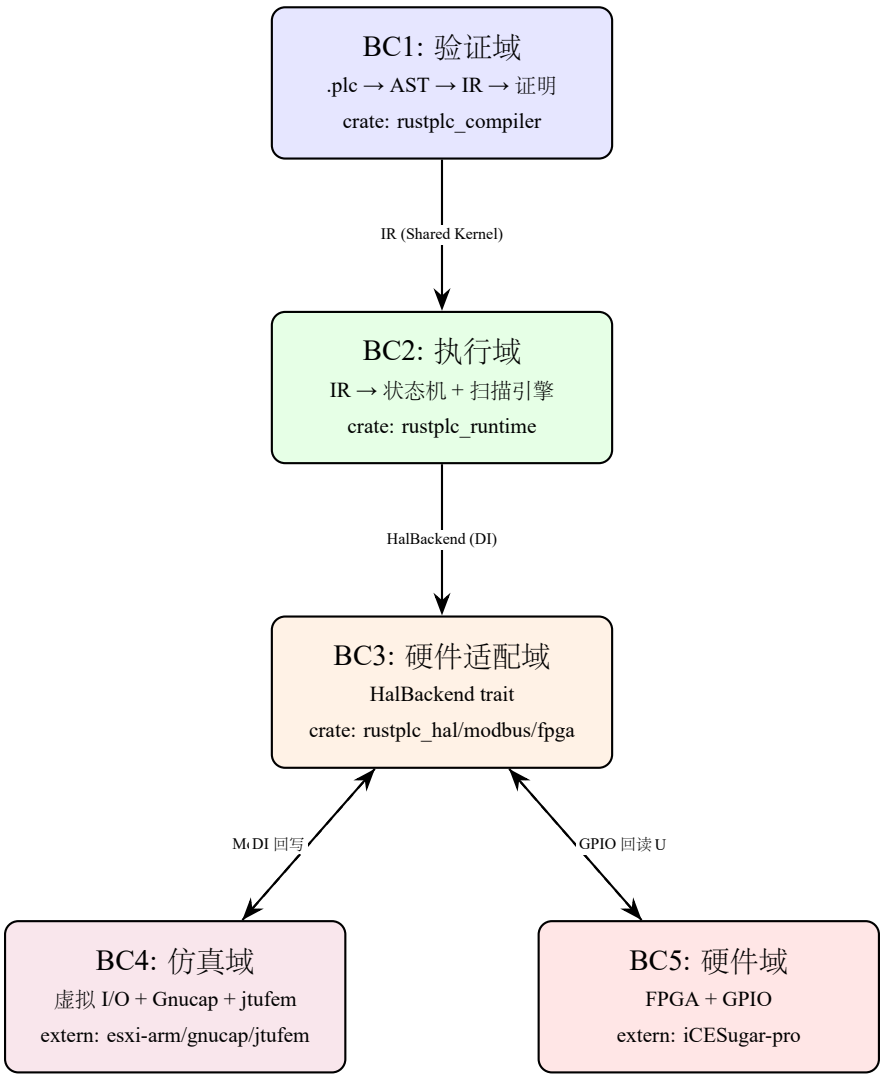


Figure 6: 五个限界上下文及其集成关系（虚线为闭环回路）

关系	模式	说明
BC1 → BC2	共享内核	IR 数据结构（`StateMachine`, `TopologyGraph`）由两个上下文共同拥有，修改需要双方同意
BC2 → BC3	依赖倒置	`HalBackend` trait 定义在 BC3，但由 BC2 的 `ScanCycleEngine` 消费
BC3 → BC4	发布语言	Modbus RTU/TCP 是公开标准协议，BC3 不需要了解 BC4 的内部实现
BC3 → BC5	发布语言	同上，Modbus RTU 帧格式是契约
BC4 → BC3	遵从者	BC4 必须按照 Modbus 寄存器映射表回写数据，不能自定义协议
BC5 → BC3	遵从者	同上，FPGA 必须实现标准 Modbus RTU slave

Table 4: 限界上下文集成模式

BC	Crate	职责
BC1	`rustplc_compiler`	解析、语义分析、四大验证引擎、代码生成
BC2	`rustplc_runtime`	`ScanCycleEngine`、`TimerBank`、生成代码的运行时支撑
BC3	`rustplc_hal`	`HalBackend` trait、`SimBackend`、`DeviceMapping`
BC3	`rustplc_modbus`	`ModbusBackend`（tokio-modbus）
BC3/BC5	`rustplc_fpga`	`FpgaBackend`（UART Modbus RTU）
跨 BC	`rustplc_orchestrator`	配置驱动的模式选择、启动编排

Table 5: 限界上下文到 Crate 的映射

Part V
仿真域 (BC4)：虚拟工厂的闭环数据流

19 虚拟工厂架构

模式 B 的核心价值是：不需要任何实体硬件，纯软件构建完整的控制闭环。这使得开发者可以在笔记本电脑上验证控制逻辑的正确性，包括物理延迟和力学响应。

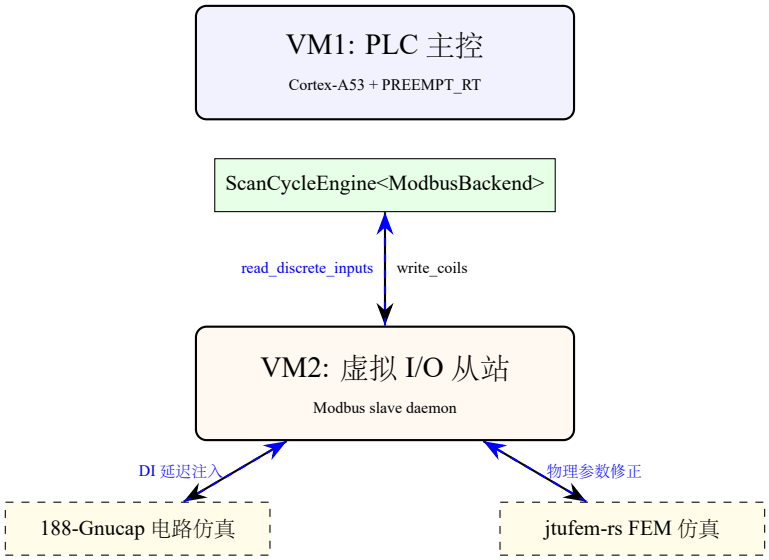


Figure 7: 模式 B 虚拟工厂数据流（蓝色虚线为闭环回路）

20 Modbus slave daemon

虚拟 I/O 从站是一个 Modbus TCP slave 进程，维护两组寄存器：

- `coils[]`——主控写入的输出信号（如电磁阀开关指令）
- `discrete_inputs[]`——从站回写的输入信号（如传感器状态）

当主控写入一个 coil 时，从站不是简单地将对应的 DI 置位，而是将写入事件转发给仿真引擎，由仿真引擎计算物理响应后再更新 DI。

21 电路仿真集成（188-Gnucap）

Gnucap 是一个开源 SPICE 电路仿真器。在虚拟工厂中，它负责模拟电气层面的物理延迟：

21.1 线圈吸合延迟

当主控写入 `stamp_valve = ON` 时，电磁阀线圈不会瞬间吸合。Gnucap 模拟 RL 电路的暂态响应：

$$i(t) = \frac{V}{R}\left(1 - e^{-\frac{R}{L}t}\right)$$

当电流达到吸合阈值 $I_{\text{pull-in}}$ 时，阀芯动作。典型的 24V DC 电磁阀吸合延迟约 15ms。

21.2 传感器信号滤波

接近传感器的输出经过 RC 低通滤波器，信号上升沿有延迟：

$$v_{\text{out}}(t) = V_{\text{cc}}\left(1 - e^{-\frac{t}{RC}}\right)$$

当 v_{out} 超过逻辑高电平阈值时，DI 才变为 true。典型延迟约 5ms。

21.3 仿真流程

1. 主控写 coil → 从站接收 Modbus 帧
2. 从站将 coil 变更事件发送给 Gnucap
3. Gnucap 运行瞬态仿真，计算延迟时间 Δt
4. 从站启动定时器，Δt 后更新对应的 DI
5. 主控在下一个扫描周期读到更新后的 DI

22 力学仿真集成（jtufem-rs）

jtufem-rs 是一个 Rust 实现的有限元方法（FEM）求解器。在虚拟工厂中，它负责模拟机械层面的物理响应：

22.1 气缸活塞杆应力分析

当气缸伸出推动工件时，活塞杆承受轴向压力。FEM 计算：

- 活塞杆应力分布（von Mises 应力）
- 最大变形量（影响冲压精度）
- 安全系数（是否超过材料屈服强度）

22.2 疲劳寿命预测

基于 S-N 曲线和 Miner 累积损伤理论，预测气缸的疲劳寿命：

$$D = \sum_{i=1}^{k} \frac{n_i}{N_i}$$

当累积损伤 $D \geq 1$ 时，预测疲劳失效。这为预测性维护提供了数据支撑。

22.3 物理参数修正

FEM 仿真的结果可以修正虚拟从站的行为参数：

- 气缸行程时间：考虑负载后，实际行程时间可能比空载时长 10–20%
- 定位精度：活塞杆变形导致工件位置偏移，影响传感器触发时机
- 振动特性：支架共振可能导致传感器误触发

Part VI
硬件域 (BC5)：FPGA 确定性 I/O

23 为什么需要 FPGA

Linux 即使使用 PREEMPT_RT 补丁，GPIO 翻转的抖动仍在 5–15μs 级别。对于大多数工业控制场景这已经足够，但某些安全关键应用需要更高的确定性：

- 紧急停机信号必须在微秒级响应
- 高速计数器（编码器脉冲）不能丢脉冲
- PWM 输出需要纳秒级精度

FPGA 的硬件并行性和确定性时序完美解决了这些问题。

24 iCESugar-pro 开发板

iCESugar-pro v1.3 基于 Lattice ECP5 FPGA，具有以下特性：

25 FPGA 内部架构

FPGA 内部实现一个硬件 Modbus RTU slave 状态机：

所有模块在硬件上并行运行：UART 收发、Modbus 帧解析、GPIO 驱动同时进行，没有操作系统调度开销。GPIO 翻转延迟 < 10ns。

参数	规格
FPGA 芯片	Lattice ECP5 LFE5U-25F
逻辑单元	24K LUT
BRAM	1008 Kbit
PMOD 接口	4 个（32 GPIO）
USB	USB-C（供电 + JTAG + UART）
工具链	Yosys + nextpnr（开源）
价格	~$50

Table 6: iCESugar-pro v1.3 规格

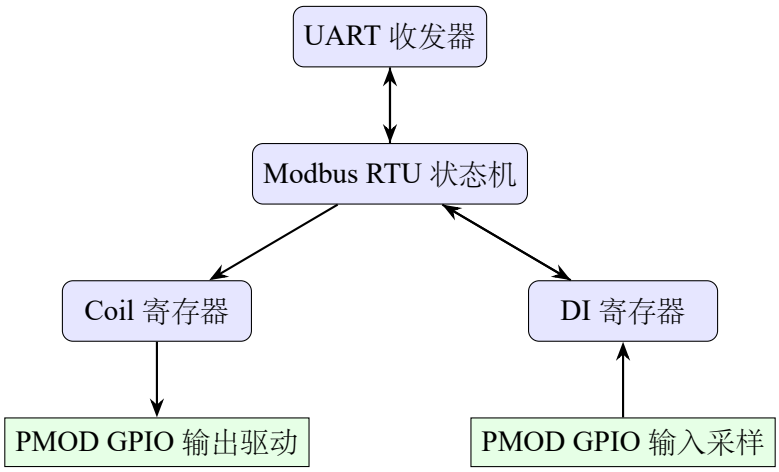


Figure 8: FPGA 内部 Modbus RTU slave 架构

PMOD 端口	外设	用途
PMOD-A[0:3]	4 路继电器模块	数字输出（电磁阀/电机）
PMOD-B[0:3]	4 路光耦隔离输入	数字输入（传感器/按钮）
PMOD-C[0:1]	UART 转 RS-485	Modbus RTU 通信
PMOD-D	预留	扩展

Table 7: PMOD 外设分配

26 PMOD 外设连接

通过 PMOD 接口连接工业级外设：

27 虚实混合模式

模式 C 的独特之处是可以混合虚拟和实体 I/O：

- 部分设备连接到 FPGA GPIO（真实继电器、真实传感器）
- 部分设备连接到虚拟从站（Gnucap 仿真、jtufem 仿真）
- 主控的 `ModbusBackend` 同时与两个从站通信

这使得开发者可以逐步将虚拟设备替换为实体设备，实现渐进式部署。

Part VII
总结与展望

28 已完成的工作

截至 2026 年 2 月，RustPLC 已完成以下里程碑：

Phase	限界上下文	核心交付物	状态
Phase 1	BC1: 验证域	DSL 解析器 + 四大验证引擎	已完成
Phase 2	BC2: 执行域	代码生成 + ScanCycleEngine + TimerBank	已完成
Phase 3	BC3: 硬件适配域	HalBackend trait + SimBackend + 配置	进行中

Table 8: 项目进度总览

28.1 验证能力

编译器已通过 59 个测试（48 单元测试 + 5 集成测试 + 6 端到端验证测试），覆盖以下验证场景：

- 安全性：双缸互斥（通过/失败）、冲压安全（通过）
- 活性：无超时等待检测（失败）、正常流程（通过）
- 时序：约束过紧检测（失败）、正常时序（通过）
- 因果：链路断裂检测（失败）、完整链路（通过）

28.2 代码生成能力

代码生成器已能将验证通过的 `.plc` 文件转换为完整的 Rust Cargo 项目，包含 `PlcState` enum、`scan_cycle` 函数和 HAL 配置。生成的代码可以用 `SimBackend` 直接运行和测试。

29 技术创新点

29.1 声明式安全

传统 PLC 编程中，安全互锁逻辑散落在梯形图的各个 rung 中，审计时需要人工追踪所有相关的触点和线圈。RustPLC 将安全约束提升为一等公民：

```
safety: stamp_head.extended conflicts_with conveyor_motor.on
```

一行声明，编译器自动证明。审计者只需检查约束是否完整，不需要理解实现细节。

29.2 物理感知编译

传统编译器不理解物理世界。RustPLC 的编译器知道：

- 电磁阀有响应时间（`response_time: 15ms`）
- 气缸有行程时间（`stroke_time: 250ms`）
- 信号沿因果链传播，每一跳都有延迟

这些物理参数直接参与时序验证和代码生成，不是注释或文档，而是编译器的输入。

29.3 三模式统一

同一份 DSL 源码、同一份生成代码，通过 `HalBackend` trait 的不同实现，可以运行在三种完全不同的环境中：

1. 纯内存仿真（CI/CD 流水线）
2. 虚拟机集群（含电路和力学仿真的数字孪生）
3. 真实硬件（FPGA + 继电器 + 传感器）

开发者在模式 A 中快速迭代，在模式 B 中验证物理行为，在模式 C 中部署到产线。整个过程不需要修改任何控制逻辑代码。

30 未来工作

30.1 Phase 3 完成：Modbus 后端

实现 `ModbusBackend`，使用 `tokio-modbus` 0.17 库，支持 Modbus RTU（串口）和 Modbus TCP（以太网）两种传输方式。这是连接虚拟工厂和实体硬件的关键桥梁。

30.2 Phase 4：虚拟 I/O 从站

实现 Modbus slave daemon，集成 Gnucap 电路仿真和 jtufem-rs 力学仿真。目标是在 196-ESXI-ARM 虚拟机集群上运行完整的数字孪生。

30.3 Phase 5：FPGA 硬件

在 iCESugar-pro 上实现 Verilog Modbus RTU slave 状态机，通过 PMOD GPIO 驱动真实的继电器和传感器。

30.4 长期愿景

- 模拟量 **I/O**：支持 4–20mA 电流环和 0–10V 电压信号，实现 PID 控制回路。
- 多控制器协同：多个 VM 运行不同的 PLC 程序，通过 Modbus 网络协同工作。
- 图形化 **DSL** 编辑器：基于 Web 的可视化编辑器，拖拽设备、连线、配置约束，自动生成 `.plc` 文件。
- **OPC UA** 集成：支持 OPC UA 协议，与主流 SCADA/HMI 系统对接。

31 成本与开源性

RustPLC 的全部技术栈 100% 开源（MIT/Apache 协议）。硬件部分（模式 C）的总成本约 $120：

硬件	用途	价格
iCESugar-pro v1.3	FPGA 开发板	~$50
PMOD 继电器模块	数字输出	~$15
PMOD 按钮/传感器	数字输入	~$15
USB-UART 适配器	Modbus RTU 通信	~$10
杜邦线/面包板	接线	~$10
合计		**~$120**

Table 9: 模式 C 硬件成本

模式 A 和模式 B 不需要任何硬件投入。一台普通笔记本电脑即可运行完整的编译验证流程和虚拟工厂仿真。

声明物理事实与安全意图，让编译器证明它是安全的；
然后生成确定性执行内核，驱动虚拟工厂或真实硬件。

RustPLC —用 Rust 写的，所以它不会 panic。
好吧，至少不会在生产线上。