2025.10 Ver0.10

ProU Quick Man系列手册

20分钟跑通和理解 NoTime（C#）

前言

NoTime是什么

NoTime（C#） 是 ProCon 平台的实时运行框架。它让用 C# 编写的普通应用程序，能够在 INtime 实时操作系统中以微秒级精度运行。

在传统自动化系统中，C# 通常只能运行在 Windows 上，受线程调度影响；而 NoTime 通过托管运行时与共享内存机制，使同一份 C# 程序逻辑在 Windows 与实时核之间自由切换。

这意味着工程师无需改变开发习惯，也能编写具备实时特性的控制程序。

为什么需要NoTime

在工业控制、半导体、精密组装等场景中，控制系统对“确定性”的要求远高于“速度”。
然而普通 Windows 线程调度无法保证 μs 级周期稳定性，即便算法正确，也可能因时间抖动导致动作不一致。

NoTime 正是为了解决这一问题。它将控制逻辑迁移至 INtime（实时系统） 实时核执行，在硬件层面实现周期调度的独立性与确定性，并通过共享内存实现 Windows 与 RTOS 间的高效同步。

工程师无需学习 RTOS，也无需修改编译流程，只需在 Visual Studio 中开发，即可获得实时性能。

关于本手册

本手册是面向初次接触 NoTime 的工程师编写的入门指南。

不需要具备实时系统背景，只要具备基本的 C# 编程经验，即可在 20 分钟内完成第一个实时例程的运行体验，并在随后的章节中逐步理解其背后的机制与原理。

从理解原理到运行示例，从共享内存通信到实时任务执行，本手册将带你快速建立对 NoTime 架构的完整认知：同一份 C# 程序，既能在 Windows 上调试，也能在实时系统中执行。

1. 文档目标与结构

1.1. Quick Start目标

理解 NoTime 的关键在于弄清楚：

“为什么一段普通的 C# 程序，在 Windows 上以线程方式运行时是不确定的，但在实时系统中却能获得 μs 级稳定周期？”

而这背后涉及托管执行模型（IL + CLR）、跨域调度、共享内存同步机制等多个概念。

许多工程师第一次接触 NoTime 时，通常会提出以下问题：

- 什么是NoTime，在什么场景下NoTime会产生价值
- NoTime的原理是什么？
- Windows 下调试的逻辑，如何切换到NoTime的 μs 周期运行？
- Windows 与实时系统之间的数据是如何同步的？
- 我现有的 Windows 程序如何迁移到 NoTime？

本 Quick Start 的目标，就是在最短时间内回答这些问题，并让你完整跑通第一个 NoTime 程序。

通过随附示例工程，你将能够在 20 分钟内理解：

- NoTime 的执行原理：IL + CLR + μs 调度
- Windows 域与实时域在 ProCon 平台中的分工
- 共享内存如何完成跨域同步
- 程序如何从 Windows 切换到实时执行
- 你的现有业务逻辑如何迁移或拆分到实时域

本章不仅帮助你“跑起来”，更重要的是让你“理解为什么能跑起来，以及两域之间是如何协作的。

1.2. 学习路径与内容结构

为了让没有实时系统背景的 C# 工程师也能快速理解 NoTime，本手册采用了“由现象到原理、由原理到工程”的结构，共分为六章与若干附录。其整体设计思路如下：

第 1 章：定位与学习目标 — 建立阅读坐标系

这一章帮助读者明确：

- 本文档要解决什么问题
- 适用于哪些读者
- 读完后能掌握什么

帮助读者建立“坐标系”。

第 2 章：ProCon 系统架构 — 理解双域协作模型

要理解 NoTime，必须先理解 ProCon 的基础结构：

- Windows 域 vs 实时域
- 两个系统为什么要分工
- 共享内存为什么存在

这章建立整个系统的“地基”，帮助你知道 NoTime 运行在怎样的系统中。

第 3 章：NoTime 的核心机制 — 解释 NoTime 为什么能工作

本章回答 NoTime 的三个根本问题：

- C# 如何跨系统运行？（IL + CLR）
- 为什么进入实时域后获得 μs 调度？（实时系统调度器）
- 跨系统同步为什么用共享内存？（可预测 / 零拷贝 / 周期友好）

这章是理解 NoTime 的“底层原理面”。

第 4 章：从 Windows 到 NoTime — 示例工程的完整运行路径

理解原理后，工程师真正关心的是：

- 两个 sln 如何协作?
- runtime.exe 是如何被加载到实时系统?
- run_in_ntf 开关如何影响执行流?
- μs 周期循环是如何被触发的?
- 共享内存在哪一层对接?

这一章完全基于随附例程，让你能从代码角度“看到 NoTime 的实际运行”。

第 5 章：从 Windows 迁移到 NoTime — 工程落地指南

大部分用户不是从零开始写，而是：“已有 Windows C# 逻辑，现在要迁移到 NoTime 实时域。”因此本章提供：

- 逻辑拆分原则（实时 vs 非实时）
- 事件驱动改为周期驱动的方法
- 实时域禁止操作列表
- 内存与线程的可预测性要求
- 迁移后的两阶段验证方法

这是对真实工程迁移最有价值的一章。

附录：深入工程细节（可按需查阅）

附录提供的是“延伸主题”，用于在掌握主线内容后进一步深入：

- 可预测多线程模型与实时线程约束
- 非共享内存通信方式的适用性
- Windows vs NoTime 的性能对比与测试方法
- 实时内存管理策略（避免 GC 与不可预测分配）
- 基于 QuickStart 的行为对比示例

附录并非入门必读，而是为评估者、架构师和需要深度理解实时系统的工程师提供更多技术细节。

使用方式建议

- 如果你是第一次接触 NoTime：建议严格按照 1 → 5 章顺序阅读一次。
- 如果你有准备迁移项目的工程目标：可以重点阅读第 4、5 章，并参考附录 B/C/D。
- 如果你是进行系统评估等架构研究：建议重点阅读第 3 章与性能对比相关的附录。

1.3. 目标读者

本手册适用于：

- 需要将 Windows 算法逻辑迁移至实时系统的自动化开发者；
- 负责验证与评估 μs 级控制方案的平台研发工程师
- 使用高级语言进行设备控制开发的软件工程师；

无需 RTOS 或 交叉编译经验，只需具备 C# 和 Visual Studio 的基础使用能力。

2. ProCon的系统架构

2.1. ProCon 系统概览

ProCon 是面向工业领域的双系统协同平台。它将通用操作系统（如Microsoft Windows ）与实时系统INtime（实时系统） 部署在同一台工业计算机上，两者并行运行、相互独立，却能通过共享内存高速协同。

INtime（实时系统）并非 Windows 的子进程，而是一个与 Windows **并行运行的独立实时内核**。两者共享同一硬件平台，但拥有各自的内存空间与调度体系。

它们通过硬件分核与共享内存通道协作，互不干扰：

特性	Windows 通用操作系统域	INtime（实时系统） 实时系统域
任务调度	时间片、动态优先级	固定优先级、μs 周期调度
内存模型	虚拟内存、分页机制	物理锁页分配
CPU 分配	可运行普通任务	可独占核心执行实时任务
通信机制	系统调用、驱动接口	共享内存 / 通信端口
容错隔离	崩溃影响用户态	实时域独立运行 ， 不受影响

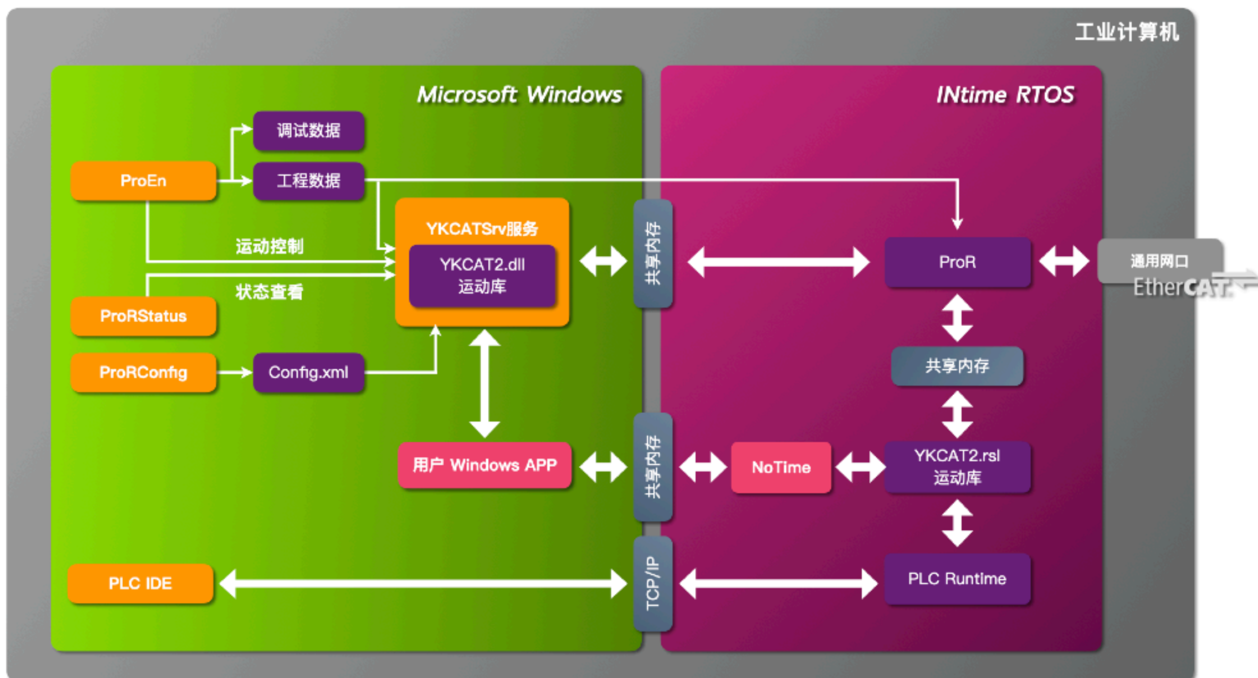
从架构上看，INtime（实时系统） 是与 Windows **并肩运行（Side-by-Side）** 的第二操作系统。它不是 Windows 的进程，也不是虚拟机，而是通过硬件分核在同一 CPU 上实现并行与确定性。

ProCon的架构如下图：

Windows（Microsoft Windows）域组件：

模块	职责说明
ProEn	工程配置与调试软件。用于生成工程数据与调试数据，通过 YKCatSrv 服务下发至实时域。
ProRStatus	状态查看工具，用于监控实时运行状态、错误码、伺服信息等。
ProRConfig	配置工具，用于生成 `Config.xml` 配置文件并提供给 YKCatSrv 服务加载。

模块	职责说明
ProRConfig	配置工具，用于生成 `Config.xml` 配置文件并提供给 YKCatSrv 服务加载。
YKCatSrv 服务	平台通信核心服务。负责管理共享内存与驱动接口，将上层应用（ProEn、ProRStatus、用户 APP）的命令统一映射到实时域。
YKCat2.dll 运动库	驱动层 API，供 Windows 应用调用，实现命令下发与状态读取。
用户 Windows APP	用户自研上位机应用，通常基于高级语言开发，直接通过 YKCat2.dll 或共享内存控制设备。
PLC IDE	可选组件，用于 PLC 程序开发与下载。通过 TCP/IP 通道与实时域中的 PLC Runtime 通信。



INtime（INtime RTOS）域组件：

模块	职责说明
ProR	实时控制运行时：负责接收命令、执行控制逻辑、规划运动曲线、运行EtherCAT协议栈
NoTime	运行在 INtime 实时域的实时执行框架，使 C#/C++ 逻辑获得 μs 级调度精度
YKCAT2.rsl	实时域对应的驱动库，与 Windows 侧的 YKCat2.dll 形成双端匹配，通过共享内存交换命令与状态数据。

模块	职责说明
PLC Runtime	可选组件，用于与 PLC 逻辑集成；通过 TCP/IP 与 Windows 侧的 PLC IDE 或 HMI 通信。
EtherCAT 接口	通过通用网口实现与现场 I/O、伺服驱动器的实时通信。

协同关系摘要：

- ProEn / ProRConfig / ProRStatus / 用户 APP → 通过 YKCatSrv + YKCat2.dll 与实时域通信；
- NoTime 通过共享内存与 YKCat2.rsl 对接，实现 μs 级执行与状态反馈；
- ProR 执行运动控制算法与EtherCAT执行协议栈，实现驱动与现场 I/O 的实时交互。

3. NoTime的核心机制

3.1. 为什么需要NoTime

在工业控制行业中，工程师大致分为两大类技术路径：

- Windows + 运动控制卡（Control Card）生态
- PLC（IEC 61131-3）生态

这两类系统解决了不同的问题，却都无法同时满足："现代软件工程能力" + "硬实时执行能力"，NoTime 的出现，就是为了解决这两类体系长期无法跨越的技术缝隙。

Windows + 运动控制卡生态的优势与限制

工业设备行业大量公司采用：

```
Windows 上位机 + 运动控制卡（PCIe / EtherCAT）
```

软件工程师喜欢它，因为：

- 能用 C#/C++ 写逻辑（最熟悉、最现代化）
- 能使用 Visual Studio、.NET、NuGet
- 能管理大型项目（Git/CI/CD）
- 可以接入各种算法框架（数学库、AI、优化）
- 调试便利（断点、日志、Profiler）
- 上位机 UI、流程逻辑都能无缝构建

这是"软件工程体验最好、开发效率最高"的控制架构

但它有一个根本弱点：高级语言写出的逻辑几乎全部运行在 Windows。

控制卡承担“实时动作”，但用户逻辑无法进入卡中

- 控制卡的特性都是：
- 实时循环在卡内
- 插补在卡内
- 状态机在卡内
- 驱动同步在卡内（如有）
- 但用户逻辑不能进入卡内（除非用 C++ 写固件）

结果是能实时的地方不能写业务逻辑（卡内），能写业务逻辑的地方不能实时（Windows）。而两者之间的连接方式一般为SDK API。这就造成巨大限制：

- 你的算法永远在 Windows → 无法 μs 周期执行
- 无法在控制卡里运行用户自定义插补/状态机
- 无法将复杂逻辑与实时动作融合
- 逻辑分裂：卡里一份（固定固件）；Windows 一份（业务逻辑）

这是高级语言 PC-based Control 自身的结构性问题。

PLC 生态的优势与局限

PLC 提供：

- 稳定的实时周期
- 与伺服/I/O 强耦合
- 工业级可靠性
- IEC 61131-3 标准语言
- 设备现场普及率高

这是“实时能力最强”的生态。

PLC 的限制本质在于：

- 语言封闭（ST/FBD/LAD）

- 不支持复杂数据结构
- 不适合构建大型软件
- 不支持现代编程范式（组件化、异步、算法库）
- 很难管理算法逻辑与平台逻辑
- 难以维护复杂的项目结构
- 无法直接运行 C#/Python/C++ 算法
- 无 Git/CI/CD/单元测试等现代工程体系

结果是：控制工程师想要实时性，但难以使用现代软件技术。

NoTime 解决的是什么?

而是解决一个根本性矛盾：让软件工程师可以用 C# 写逻辑，并让这段逻辑以 μs 级周期在实时系统中执行。

这意味着你写的 C# 逻辑：

- 在 Windows 下可调试
- 在实时系统下获得 μs 调度
- 与伺服 / EtherCAT / I/O 同步
- 获得跟 PLC 一样甚至更强的实时性

第一次实现：“现代软件开发方式” + “工业级实时执行能力”同时成立。

3.2. C# 为何无需交叉编译即可进入实时系统（托管执行 + μs 调度）

在理解 NoTime 之前，首先要回答一个关键问题：为什么同一段 C# 代码既能在 Windows 下运行，又能在 INtime（实时系统）中执行?

这看上去像“跨平台编译”，但实际完全不是。

原因来自 C# 的语言结构与运行模型：C# 并非直接编译成机器码，而是采用托管执行（Managed Execution）。

这一点与 C/C++、PLC 固件、运动控制卡固件完全不同。

C# 编译为 IL：与平台解耦的根本原因

C# 程序编译后生成的不是 CPU 机器码，而是：

- IL（Intermediate Language，中间语言）
- 元数据（Metadata）
- 类型与签名（Type / Method Signature）

IL 有三个关键特性：

- 与 CPU 架构无关：（不依赖 x86 / x64 / ARM）
- 与操作系统无关：（Windows / Linux / RTOS 都可加载）
- 不需要交叉编译工具链

因此，C# 的构建结果并不是一个“特定平台的可执行文件”，而是一份可由任意托管执行环境解释或编译的中间格式。

CLR：决定 IL 如何执行，而不是操作系统本身

IL 并不能直接运行，它必须由托管执行环境加载和执行。

这个环境就是：CLR（Common Language Runtime，公共语言运行时）

CLR 负责：

加载并验证 IL

管理托管内存

执行 IL（解释或编译成本地机器码）

调用托管线程

管理异常

反射与元数据解析

因此：决定 C# 是否能运行的不是操作系统，而是是否存在一个 CLR。有 CLR 的地方，IL 就能跑。IL + CLR 赋予 C# 强大的跨系统可移植性。

NoTime（C#） 并不是“跨编译框架”，而是“跨系统调度框架”

传统实时系统不能执行 C#，原因很简单：

- 它们没有 CLR
- 它们不支持 IL
- 它们只能运行固件级机器码

而 NoTime （C#）：在实时系统中提供托管执行环境CLR（可加载 IL）， 由INtime 的 μs 调度器驱动这段托管执行逻辑。也就是说：NoTime 不改变你的代码，而是改变“谁来调度你的代码”。这与 PLC、运动控制卡、NoTime（C++）完全不同。

Windows 域：CLR 由 Windows 调度器驱动

在 Windows 域下执行 C#：

- 线程调度由 OS 决定
- 抢占、上下文切换、GC 都不可预测
- Jitter 通常在毫秒级

因此：

- Windows 域适合开发 / 调试
- 不适合“精确定时执行”

这就是为什么 Windows 域是 C# 工程师的“开发场”，不是实时执行场。

在 INtime（实时系统）中运行：CLR 被 μs 调度器驱动

INtime 提供：

- μs 级固定周期
- 硬实时调度保证
- 抖动可控

- 精确的周期窗口

在 NoTime（C#） 中：

- CLR 的执行时机由 μs 调度器决定，不是由 Windows 线程调度器决定
- C# 逻辑获得实时性，尽管 IL 与 Windows 下相同，但调度机制完全改变：
    - 时间行为固定
    - 延迟可预测
    - 抖动极低
    - 与伺服/I/O 的同步得以实现

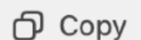
这就是 NoTime（C#） 的核心特性：同一段 C#，在不同调度环境下呈现完全不同的时间行为。

为什么这一切意味着不需要交叉编译?

因为：

- C# 不输出机器码 → 无需根据平台重新编译
- IL 可跨平台移动
- CLR 在 Windows 和实时系统中都能运行
- JIT （Just-In-Time Compilation，即使编译）根据当前 CPU 自动生成目标机器码

因此：



```
开发(Windows)
↓
编译为 IL
↓
相同 IL 载入实时系统(NoTime 托管)
↓
JIT → 在实时域生成本地机器码
↓
μs 调度器周期执行
```

一句话总结：C# 能在实时系统运行不是因为它“被重新编译”，而是因为实时系统能够“调度并执行它”。

小结：C# 为什么可以直接进入实时系统?

因为：

- IL 是跨系统格式 → 不绑定 CPU/OS
- CLR 能解释/JIT IL → 任何有 CLR 的地方都可以执行
- JIT 将 IL 转成本地机器码 → 无需交叉编译
- INtime μs 调度器驱动 CLR → 实时性来自执行环境

因此：NoTime 的价值不在于改变 C#，而在于改变 C# 的“执行者”。

3.3. 双域协同模型：Windows 与实时域如何分工

在前两节中，我们分别介绍了：

- C# 为什么能跨系统运行（托管执行）
- 为什么能获得 μs 级实时性（RT 调度）

接下来的问题是：

为什么 NoTime 要把系统分成“Windows 域 + 实时域”两部分?

为什么不能所有逻辑都跑在一个系统里?

为什么不能只用 Windows?

Windows 的优势非常明显：

- 强大的开发生态
- 丰富的 UI 和工具链
- 完整的文件 / 网络能力
- 易于集成数据库、算法框架
- 工程师熟悉 C# 的开发方式

- 方便调试和部署

但 Windows 的调度模型是：

- 抢占式调度（preemptive）
- 多任务竞争 CPU
- 执行时机不确定
- 抖动可能从微秒到毫秒到几十毫秒
- 无法与伺服总线精确定步

因此：Windows 适合“工程逻辑 + 人机界面”，不适合“精准时间控制”。这不是 Windows 的缺点，而是它的设计目标决定的。

为什么不能只用实时系统？

实时系统（如 INtime RTOS）的优势是：

- μs 级固定周期
- 抖动极低
- 与 I/O 与 EtherCAT 同步
- 可预测的调度行为

但是实时系统不适合承载：

- 大量 UI、图形渲染
- 文件系统、大内存堆
- 复杂的网络协议栈
- 大规模软件框架
- 数据库、云连接
- 高层调度、业务流程逻辑
- 复杂的工程工具链

更关键的是：实时系统应保持最小化、确定性和可预测性。如果把所有业务逻辑都塞进实时系统，会导致：

- 调度复杂化
- 资源抢占影响实时性
- 工程规模难以扩展
- 难以维护
- 依赖特定 RTOS，不利于软件工程

双域模型：各司其职

因此，在现代工业控制架构中，最佳实践是：

```
Windows 域（管理域）
          +
实时域（执行域）
```

这种“双域模型”是经过多年工业实践检验的结构。

- Windows 域负责（也可以是其他通用操作系统，如Linux）：
- 工程配置
- UI / 参数 / 任务控制
- 文件读写
- 高层逻辑（流程、策略）
- 调试、日志、可视化
- 启动/停止、系统状态管理
- 与外部系统（MES、云平台、数据库）交互

总结：

Windows 域 = 工程能力 + 生态能力

实时域负责：

- 周期任务（μs 定时）
- 运动控制 / 插补
- 实时 I/O
- 与伺服系统同步
- 时间敏感算法
- 生产节拍相关的核心逻辑

总结：

实时域 = 时间确定性 + 执行稳定性

两者在职责上是互补的。

NoTime 在双域模型中的位置

开发域（Windows开发域）

- 工程师在 Windows 上使用 Visual Studio 编写 C#
- 代码编译为 IL（平台无关的中间语言）
- Windows 通过托管运行时执行 IL
- 用于调试、仿真、配置与工程管理

执行域（实时域 INtime）

- 相同的 IL 被实时系统中的托管运行时重新加载
- 实时域在加载 IL 后进行独立的 JIT，生成本地机器码
- 机器码由 INtime 的 μs 级调度器驱动执行
- 周期逻辑、动作控制、伺服同步全部在实时系统中完成
- 不依赖 Windows 的运行时、线程调度或时间行为

结论

- 开发在 Windows，实时执行在 INtime。
- 相同的 IL 在 Windows 和实时系统中分别由各自的托管运行时加载并执行
- 实时行为完全由实时系统决定。

换句话说，NoTime 让 C# 的开发与执行在两个域中自然分工：开发在 Windows，实时执行在 INtime。这里的重点不是“代码写在 Windows”，而是：不需要为实时系统单独开发另一套语言或工程环境。同一套 C# 逻辑可以自然地在实时系统中执行。这是传统 PC-based 控制或 PLC 控制无法做到的。

3.4. 跨系统同步与共享内存模型：为什么 NoTime 采用共享内存

在双域结构中：

- Windows 域（工程域）负责参数、配置、任务控制、流程逻辑、可视化；
- 实时域（INtime）负责 μs 级周期逻辑、运动控制、I/O 同步与实时状态机。

两个域彼此独立运行，但必须持续交换状态与指令。因此 NoTime 必须提供一种跨系统的数据同步机制。

为什么双域架构必须进行状态同步

Windows 与实时域各自承担不同职责：

- Windows 域需要向实时域传递：
- 运行/停止指令
- 参数修改
- 任务切换
- 上层流程状态

实时域需要向 Windows 回传：

- 当前运行状态

- 周期数据
- 报警与错误码
- 实际执行反馈

如果没有稳定的同步机制：

- UI 数据无法反映实时执行情况
- 参数更新无法在实时域中生效
- 流程逻辑和动作执行会脱节
- 开发者无法调试与监控实时行为

因此在双域结构中：状态同步不是可选项，而是必须项。

为什么共享内存是实现跨域同步的最佳方式

共享内存（Shared Memory）的本质是：两个系统直接访问同一段物理内存，不经过 OS 通信机制。这带来几个关键优势：

这带来几个关键优势：

① 零拷贝（Zero Copy）

数据不需要序列化、不需要复制、不需要排队。

```
写方：直接写 RAM
读方：直接读 RAM
```

访问延迟就是内存访问延迟（纳秒级）

② 无系统调用（No System Call）

这意味着：不会受到操作系统调度影响。对于实时系统，这一点极其重要。

③ 延迟与抖动可预测

共享内存访问属于：

- 固定地址读取

- 不产生阻塞
- 不依赖第三方资源

完全符合实时控制“可预测、可重复”的要求。

④ 适用于周期性访问（Real–Time Loop Friendly）

实时循环中（例如 100μs 周期），需要：

- 在固定时间读取命令
- 在固定时间写回状态

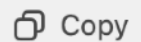
共享内存的访问开销足够小，适合放在实时循环内部执行。

自然满足“状态机（State Machine）”模型

⑤ 实时控制依赖大量状态位：

- Mode
- Run/Stop
- Fault
- Step
- AxisState
- CommandCode

共享内存非常适合做“状态镜像”：




```
Windows 写入新的指令状态
实时域周期读取并执行

实时域写回当前运行状态
Windows 周期 / 事件式读取并显示
```

这是工业控制最自然的结构。

共享内存的数据完整性保证机制

在跨域协同中，共享内存（Shared Memory）不仅承担 Windows 与实时系统之间的高速同步角色，更需要确保在高频读写（如 100 µs 周期）下数据始终保持一致和有效。本节介绍共享内存的数据完整性来源，并解释为什么 NoTime（C#）能够在不使用锁（mutex）、不使用消息队列的情况下仍保持安全可靠的数据交换。

共享内存的数据完整性由 系统层、硬件层、应用层 三部分共同保证。

① 系统层：INtime 实时系统的原子写入机制

在实时域（INtime RTOS）中，NoTime 使用由 RTOS 分配和管理的固定物理内存页作为共享区。该内存具备：

- 单写多读模型（Single Writer, Multi-Reader）
- 内存页锁定（Page-Locked Memory）
- 结构体整体写入
- 不会出现部分写入（Partial Write）或撕裂（Tearing）

这意味着：

无论是 Windows 侧写入还是实时侧写入，共享内存中的字段都以“整体原子化”的方式更新，不会在写入中途被另一个进程读到“半套数据”。这是数据一致性的第一道保障。

② 硬件层：CPU 对基本类型的原子写支持

共享内存结构体中所有字段均为：

- int / uint（32 bit）
- double（64 bit）
- enum / bool（通常编译为 32 bit）

在 x86/x64 处理器架构上，这些类型均属于 原子写类型（Atomic Write Types）。

这意味着：

- CPU 一次写入整个字段（32/64 bit）
- 不会被中断拆分为多次写操作
- 第二个线程无法读取到“写入到一半的值”

因此，在硬件层无需加锁即可保证字段级别的完整性。

③ 应用层：控制字段机制（Control Flags）确保“数据帧”的完整性

尽管系统层与硬件层确保了字段不会“写一半被读”，

但我们仍需确保“整套参数”在写入完成之前不会被实时侧提前读取。

为此，应用层使用一种简单可靠的数据帧有效性策略（Frame Validity Strategy）：

写入侧（Windows）遵循的原则：

- 先写入一整套参数字段（位置、速度、模式、轴号等实际业务数据）
- 最后写入一个控制字段（Control Flag）表示“本周期的数据已经写完，可以使用”。

读取侧（实时系统）的规则：

- 只有在检测到控制字段处于“有效”状态时才会读取并使用本周期的数据
- 在此之前，所有参数均被视为“尚未写完”

执行结束后，读取侧会更新控制字段为“已处理”或“无效”

这样形成了完整的闭环：

- “先写数据 → 后写有效标志”
- “先读有效标志 → 再读数据内容”

结果：共享内存中不会出现“参数写到一半、实时系统就提前读取”的情况。这是应用层保障数据一致性的关键机制，也是工业实时系统中常用的写法。

数据完整性框图（示意图）：

3.5. 小结：支撑 NoTime（C#）跨域执行的四个关键机制

本章从语言特性、运行时机制与实时系统调度三个维度解释了“为什么一段普通的 C# 程序可以在实时系统中执行”。核心要点可总结为以下四点：

IL（中间语言）是跨平台格式，天然可跨系统移动

- C# 编译生成 IL，它与 CPU 架构、操作系统无关。同一份 IL 可同时被 Windows 的 CLR 与实时域的托管运行时加载，这使得代码本身无需交叉编译。

CLR（托管运行时）决定 IL 的执行者，而不是操作系统本身

- 在 Windows 下，CLR 被 Windows 调度；在 INtime 实时域中，CLR 被 μs 级调度器驱动。逻辑不变，但“谁来调度”完全改变了代码的时间行为。

μs 级实时调度让同一份 C# 逻辑获得可预测的执行时间

- 进入实时域后，周期循环由 INtime 的 μs 调度器固定触发，执行时机、抖动、延迟都变得可预测，这正是 Windows域无法提供的能力。

共享内存提供了跨域同步的高速通道

- Windows 与实时域拥有各自的线程体系，但通过共享内存共享一份固定布局的数据结构。它具备零拷贝、固定延迟、适合周期访问等特性，是 NoTime 的默认同步方式。

4. 从 Windows 到 NoTime：程序运行流程与入口关系

前几章的内容帮助我们从原理层面回答了一个问题：“为什么一段普通的 C# 程序，能够在实时系统中执行？”

从本章开始，我们进入实践部分。这部分不再讨论原理，而是带你沿着教学例程的源码，完整走一遍程序的运行路径。

我们将看到：

- Windows 侧工程如何初始化与监控；
- Runtime 工程（实时系统）如何加载并执行主循环；
- 共享内存在两侧的连接点；
- 程序如何从“线程调度”切换为“μs调度”。

这一章的目标是让你顺着代码就能理解系统行为：看到它怎么启动、怎么循环、怎么同步，而不仅仅是它“为什么能做到”。请在阅读时打开示例工程 Windows.sln和Runtime.sln一边看文件结构，一边对照本章内容。

4.1. Windows.sln的程序主结构

Windows 工程是整个 QuickStart 例程的启动端和监控端。其主要职责包括：

- 打开/映射共享内存
- 向共享内存写入一次性参数
- 启动实时程序（当运行模式为实时模式）

- 读取实时程序写回的状态与结果

示例工程的目录结构如下：

```text
Windows.sln
└─ Windows
    ├─ Windows.csproj
    ├─ Program.cs
    ├─ Worker.cs
    └─ Properties
```

关键说明

- Windows 工程不包含 QuickStart.cs，QuickStart 是实时域（Runtime）中的控制例程。
- Windows 不执行实时循环（不包含 μs 调度逻辑）Windows 的 Worker.cs 只做“写参数 + 读结果”。
- MemOper.cs 在 Common 目录中，保证两侧的数据结构一致。

4.2. Runtime.sln 的程序主结构

- Runtime 工程运行在 INtime（实时系统）中，是整个 QuickStart 示例的实时执行核心。其主要工作包括：
- 创建共享内存
- 启动实时线程
- 进入由 μs 调度器触发的固定周期循环
- 执行 QuickStart 示例逻辑
- 将执行状态写回共享内存

目录结构如下：

关键说明

```text
Runtime.sln
└─ Runtime
    ├─ Runtime.csproj
    ├─ Program.cs
    ├─ Worker.cs
    ├─ QuickStart.cs
    └─ Properties
```

- QuickStart 是实际控制例程
- Worker.cs 内部的周期循环由 INtime μs 调度器触发，而非 Windows 线程
- 共享内存结构定义由 Common/MemOper.cs 提供，与 Windows 工程保持一致

4.3. 运行模式开关（run_in_ntf）

示例工程通过 MemOper.cs 中的：

```csharp
public static NOS_BOOL run_in_ntf;
```

值	执行位置	调度方式	说明
NOS_FALSE	Windows 模式	普通线程调度	调试模式（runtime.exe 在 Windows 运行）
NOS_TRUE	实时模式	μs 级周期调度	正式运行（runtime.exe 在实时域运行）

当 run_in_ntf 为 false 时，程序在 Windows 工程中执行主循环 (Worker.Run())，此时你可以直接在 Visual Studio 中调试。当其设置为 true 时，Windows 工程仅初始化共享内存和状态，主逻辑会交由 Runtime 工程执行（由 μs 调度器控制）。

下面分别说明两种模式。

run_in_ntf = false（调试模式）

此模式用于阅读例程、调试逻辑、观察输出。

流程：

- Visual Studio 运行 Runtime.sln → runtime.exe 在 Windows 上运行
- Visual Studio 运行 Windows.sln
- Windows.exe 使用共享内存驱动 runtime.exe

特点：

- 所有代码均在 Windows 环境运行
- 循环由 Windows 线程调度执行
- 时间行为非确定性（抖动可见）
- 非常适合调试 QuickStart 状态机逻辑

run_in_ntf = true（实时模式）

此模式用于真实的 μs 级周期执行。

流程：

- Visual Studio 仅编译 Runtime.sln，不运行
- 运行 Windows.exe
- Windows.exe 调用：

```
ProCon.YKM_LoadNTFDotNet(runtime.exe)
```

将 runtime.exe 加载到 INtime（实时系统）

4. runtime.exe 进入实时域，由 μs 调度器触发周期循环

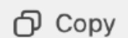
5. Windows.exe 只负责写入参数与读取状态

特点：

- 逻辑运行在 INtime 实时内核
- 执行周期稳定、无抖动
- 适用于真实设备运行与测试

4.4. 调用链与执行关系

Windows 侧调用链

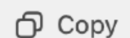


```
Program → MemOper.OpenShareMemory → Worker.StartTask
```

Windows 负责：

- 写入一次性参数（如位置/速度/模式等）
- 写入控制字段（表示“数据已准备好”）
- 循环检查实时域的反馈字段

Runtime 侧调用链（INtime 实时系统）



```
Program → Worker.StartThread → Worker.RunLoop → QuickStart.Loop
```

Runtime 负责：

- 周期性读取共享内存中的参数
- 执行 QuickStart 的状态机控制逻辑
- 将状态与结果写回共享内存

该循环由 INtime μs 调度器以固定周期触发，而非 Windows 线程。

这一节介绍了 Windows 与 Runtime 的调用链和数据流动方式。下一节将基于示例工程的实际代码，介绍共享内存如何在两个域之间读写，从而支撑这一调用链。

4.5. 共享内存的实际使用方式

在本示例中，Windows 与 Runtime 之间的共享内存由 ProCon / NoTime 平台通过 NoSys / YKCat2 封装好，用户代码通过 Common/MemOper.cs 中的 MemConfig 和 QuickStartCFG 这两个结构体，与共享内存进行读写。

整体思路可以概括为：Runtime 负责创建共享内存块，Windows 负责打开同一块共享内存，两侧通过 MemOper.memConfig->quickStartCFG 这一段内存交换命令与状态。下面分步骤说明：

共享内存布局：MemConfig + QuickStartCFG

在 Common/MemOper.cs 中定义了一个 unsafe struct MemConfig，其中包含一块通用区（例如测试用数组、退出命令等），以及一个专门给 QuickStart 用的配置区：

csharp Copy
```csharp
unsafe public struct MemConfig
{
    public int smTest;
    public int exit;
    public fixed int array[256];

    // 快速入门例程的共享区
    public QuickStartCFG quickStartCFG;
}
```

QuickStartCFG 则是 QuickStart 例程真正关心的数据结构，包括：

- 启动标志：enable
- 状态机阶段：phase
- 轴号：axis_index
- 输出点/输入点索引：output_byte_index / output_bit_index / input_byte_index / input_bit_index
- 位置参数：position_a / position_b

- 完成标志：done

共享内存的指针在 MemOper 里以静态指针的形式保存：

csharp

```csharp
unsafe public static MemConfig* memConfig { get; private set; }
```

所有后续对共享内存的访问，都是以 MemOper.memConfig 为入口。

Runtime 侧：创建共享内存并挂接 MemConfig 指针

在 Runtime/Program.cs 中，Main 函数在加载完运动库之后，会调用 MemOper.CreateShareMemory() 创建共享内存：

csharp

```csharp
NOS_RESULT_CODE nos_result_code = MemOper.CreateShareMemory();
if (nos_result_code != NOS_RESULT_CODE.NOS_RET_OK)
{
    Console.WriteLine($"CreateShareMemory 失败({nos_result_code})");
    return;
}
```

CreateShareMemory 的实现也在 MemOper.cs 中，大致逻辑为：

- 通过 typeof(MemConfig) 获取结构体大小。
- 调用 NoSys.NOS_InitShareMemory 进行初始化。
- 调用 NoSys.NOS_CreateShareMemory(memName, size, out addr) 创建共享内存块。
- 将返回地址 addr 转换为 MemConfig*，赋给 memConfig 静态指针。

因此，从 Runtime 的角度看：只要 CreateShareMemory 成功，MemOper.memConfig 就指向了一块跨域共享的物理内存。

Windows 侧：打开同一块共享内存

在 Windows 侧的 Windows/Program.cs 中，在确认 Runtime 应用已经加载到 NoTime 后，会通过 MemOper.OpenShareMemory 打开同一块共享内存：

```csharp
// --------------打开共享内存--------------
while (true)
{
    NOS_RESULT_CODE nos_result = MemOper.OpenShareMemory(NOS_NODE.NOS_ECAT_A);
    if (nos_result == NOS_RESULT_CODE.NOS_RET_OK)
        break;

    Thread.Sleep(300);
}
```

OpenShareMemory 的实现同样在 MemOper.cs：

- 调用 NoSys.NOS_OpenShareMemory(node, RTA_NAME, MEM_NAME, out addr, run_in_ntf)
- 将返回的地址转换为 MemConfig*，赋给 MemOper.memConfig。

这样，Windows 和 Runtime 就通过同一个 MemConfig 结构体，指向了同一块物理内存，实现了跨系统的数据镜像。

Windows 侧：写入 QuickStart 命令（QuickStartExample）

在 Windows/Worker.cs 中，QuickStart 例程的启动逻辑大致如下：

① 调用 InitQuickStart() 完成参数初始化（轴号、IO 索引、位置 A/B 等）：

```csharp
unsafe private static bool InitQuickStart()
{
    // 获取轴列表
    uint axis_num = 0;
    uint[] axis_list = new uint[(int)YKE_SYSTEM_DEFINE.YKE_AXIS_NUM];
    var result = ProCon.YKM_GetAxisList(0, out axis_num, axis_list, axis_list.Length);
    if (result != YKE_RESULT_CODE.YKE_RET_OK || axis_num == 0)
        return false;

    QuickStartCFG* quick_start = &MemOper.memConfig->quickStartCFG;
    quick_start->axis_index = axis_list[0];
    quick_start->output_byte_index = 0;
    quick_start->output_bit_index = 0;
    quick_start->input_byte_index = 0;
    quick_start->input_bit_index = 0;
    quick_start->position_a = 100.0;
    quick_start->position_b = 200.0;
    return true;
}
```

② 启动 QuickStart 任务：

csharp Copy

```csharp
unsafe private static void QuickStartExample()
{
    if (InitQuickStart() == false)
        return;

    QuickStartCFG* quick_start = &MemOper.memConfig->quickStartCFG;
    quick_start->done    = YKE_BOOL.YKE_FALSE;
    quick_start->enable = YKE_BOOL.YKE_TRUE;

    while (!m_cancel)
    {
        // Windows 自己的任务（如 UI 刷新等）
        System.Threading.Thread.Sleep(1);

        // 等待 Runtime 把 done 置为 TRUE
        if (YKE_BOOL.YKE_TRUE == quick_start->done)
            break;
    }
}
```

可以看到：

- Windows 侧只在启动时写一次 enable = TRUE 和参数；
- 然后以很低的频率轮询 done 标志；
- 真正的周期控制逻辑不在 Windows，而是在 Runtime。

Runtime 侧：周期调用 QuickStart 状态机

在 Runtime/Worker.cs 中，启动线程后，会在一个循环中反复调用 QuickStart：

csharp Copy

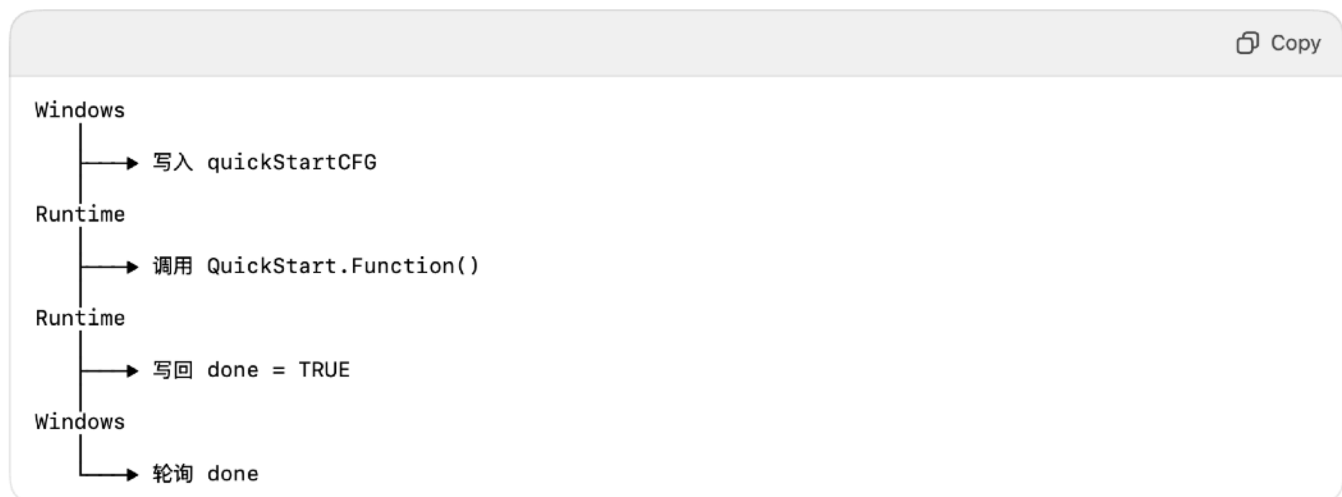
```csharp
public static void StartThead()
{
    var worker = new System.Threading.Thread(ThreadTask);
    worker.Start();
}

unsafe private static void ThreadTask()
{
    while (!mCancel)
    {
        System.Threading.Thread.Sleep(1);

        // 示例数组计数
        MemOper.IncArray(0);

        // 核心：周期调用 QuickStart 状态机
        QuickStart.Function(&MemOper.memConfig->quickStartCFG);
    }
}
```

QuickStart.Function 内部就是你在 QuickStart.cs 中实现的状态机逻辑，典型流程是：

- 检查 enable 是否为 TRUE，若为 TRUE 则推进 phase。
- phase = 10：启动运动到 A 点、输出置 ON。
- phase = 20：等待到达 A 点。
- phase = 30：启动运动到 B 点、输出置 OFF。
- phase = 40：等待到达 B 点，然后把 done 置为 TRUE。

这整个过程中，QuickStart 始终通过 config 指针访问共享内存中的 QuickStartCFG 字段。

换句话说，Runtime 周期性地“消费” Windows 写入的命令，并把执行结果（phase、done 等）写回同一块共享内存。



数据流总结

结合上一节的 4.4 调用链描述，可以把本节的共享内存读写关系总结为：

- Runtime 启动阶段：MemOper.CreateShareMemory() 创建共享内存，并将 MemOper.memConfig 指向这块内存。
- Windows 启动阶段：MemOper.OpenShareMemory() 打开同一块共享内存，同样获得 MemOper.memConfig 指针。

- Windows 写入启动命令：InitQuickStart() 填充 quickStartCFG 里的参数，将 enable 置 TRUE，done 置 FALSE。
- Runtime 周期执行：线程循环中反复调用 QuickStart.Function(&MemOper.memConfig->quickStartCFG)。根据 enable/phase 推进状态机，并在动作完成后将 done 置为 TRUE。
- Windows 轮询结束条件：Windows 线程中轮询 quick_start->done，检测到 TRUE 后退出。

4.6. 执行过程概览

整个示例工程的执行过程可以分为六个阶段：

① Windows.exe 启动并打开共享内存

- 初始化 MemOper
- 创建或映射共享区
- 准备写入参数

② 判断运行模式（run_in_ntf）

- run_in_ntf = false → runtime.exe 作为普通 Windows 进程运行
- run_in_ntf = true → runtime.exe 将由 Windows 加载到实时系统

③ runtime.exe（实时系统）启动

- 初始化共享内存指针
- 创建实时线程
- 进入 Worker.RunLoop()（固定周期）

④ Windows 向共享内存写入一帧参数

包含：

- 业务参数（如位置、速度、模式等）
- 最后写入控制字段（表示“参数有效，可以读取”）

⑤ Runtime 周期读取参数并执行 QuickStart

进入 QuickStart.Loop()，以固定周期推进状态机。

每周期写回：

当前阶段

执行状态

已处理标志

⑥ Windows 检测到完成标志

Windows.exe 通过轮询共享内存判断：

- 本周期是否已执行
- 状态机是否已完成

至此，本次动作流程结束。

4.7. 两种运行模式的行为差异

项目	Windows 模式	实时模式
运行位置	Windows CLR	NoTime 托管运行时
调度方式	Windows 线程调度	μs 周期调度
时间稳定性	受系统负载影响	完全确定性
状态机执行节奏	不均匀	固定周期
输出表现	有抖动	平滑稳定
适用场景	调试、验证逻辑	实际设备运行

4.8. NoTime开发者理解要点

在使用 NoTime 编写实时控制逻辑时，开发者需要理解实时系统与 Windows 在职责上的根本差异，并遵守一套严格的实时编程规范。本节总结实际工程中必须遵守的关键点。

① Windows 与实时系统的职责分工

Windows（开发/监控侧）负责：

- 配置、参数输入、UI
- 文件与网络操作
- 数据记录、报警、人机交互
- 启动和监控实时系统

实时系统（INtime）负责：

- μs 级周期控制
- 状态机推进
- 与 EtherCAT/I/O 同步的动作逻辑
- 所有“时间敏感”的控制流程

这一分工是系统架构稳定性的基础。

② 周期循环（RunLoop）必须遵守的实时编程规范

实时循环具有严格的时间预算，因此必须遵循以下规则：

禁止动态内存分配（Allocation–Free）

周期中禁止：

- new 对象
- 创建 List 或动态扩容容器
- 触发 GC（垃圾回收）

动态分配将导致不可预测的延迟，影响周期稳定性。

禁止等待类操作（Non–Blocking Only）

周期中禁止：

- Thread.Sleep()
- await / 异步等待

- Task.Delay()
- lock 竞争
- 网络 IO
- 文件 IO

周期必须是“确定性执行 + 确定性退出”。

禁止创建自由线程

实时任务只能使用 INtime 的实时线程（通过 NoTime 内部管理）。

用户不能在周期中创建普通 C# 线程。

③ 共享内存（Shared Memory）的使用原则

共享内存是状态镜像，不是通信协议。

使用方式：

- 周期读取一次（快照）
- 周期写回一次
- 所有字段必须是固定长度基本类型
- 结构必须在 Windows 与 Runtime 中完全一致
- 字段顺序不可改变

错误方式：

- 在共享内存写“事件触发”
- 在运行过程中修改结构体布局

④ 业务逻辑应该如何分层放置

放入实时域（Runtime）的：

- 对时间敏感的动作逻辑
- μs 周期状态机推进
- 跟 I/O / EtherCAT 同步的控制命令

- 运行中必须严格按周期执行的计算

放在 Windows 的：

- UI、参数设置、配方管理
- 文件系统、网络、数据库
- 大量计算但不要求立即执行的逻辑
- 控制台输出、日志写入

原则：动作放在实时系统，管理放在 Windows。

4.8. 从 Windows 迁移到 NoTime 的核心要点

本节汇总了用户在将现有 Windows C# 程序迁移到 NoTime（INtime）实时系统时必须了解的关键点。

这些要点基于实际迁移过程中反复出现的问题整理而成，旨在帮助开发者顺利完成架构改造，避免将不适合进入实时系统的逻辑直接搬入 NoTime。

实时逻辑与非实时逻辑的拆分原则

迁移的第一步，是区分哪些逻辑必须进入实时域，哪些必须保留在 Windows 侧。

建议迁入 NoTime 的逻辑（时间敏感）：

- 控制循环与动作控制
- 阶段/步骤状态机
- 与 EtherCAT/I/O 相关的同步逻辑
- 对周期（如 125 μs、250 μs）有明确要求的控制运算

必须留在 Windows 的逻辑（非时间敏感）：

- UI 与图形界面

- 日志记录
- 配方、参数管理
- 网络通信（TCP/HTTP/MQTT 等）
- 文件访问、数据库访问
- 外部系统交互（MES、ERP、云平台）

这种分层能确保实时域保持轻量且可靠，Windows 负责复杂但不敏感的部分。

周期执行模型与状态机化改造

与 Windows 事件驱动、阻塞式编程模式不同，NoTime 的实时逻辑基于：

- 固定周期调度（μs 级）
- 每个周期执行“小步推进”
- 通过状态机管理执行流程
- 不允许阻塞等待

因此，迁移旧代码时需要：

- 将阻塞式逻辑（等待某条件发生）改为周期检查
- 将“长任务”拆成多个阶段，通过状态机推进
- 移除 while(true) + sleep/wait 结构
- 保证每个周期均可在时间预算内完成

这种转换是从“事件驱动”向“周期驱动”的根本性改造。

实时域禁止操作列表

为确保实时周期的稳定性，以下操作严禁放入实时循环中：

- 文件 IO（读写日志、配置文件等）
- 网络 IO（TCP/HTTP/Socket）
- 串口、USB 等慢速外设通信
- Console.WriteLine 等大量输出

- sleep / wait / lock 等阻塞操作
- 动态内存分配（new、容器扩容）
- 复杂 LINQ / 字符串拼接（会产生临时对象）
- 异步/Task 或线程池调度

这类操作不具备时间可预测性，会导致即时的周期失效或延迟抖动。

实时域内的内存管理原则（概要）

为了保证实时执行时间具有可预测性，需要遵守以下规则：

- 实时循环中禁止动态分配内存（包括 new 数组、new 对象）
- 所有需要的大内存必须在初始化阶段一次性分配
- 运行中仅复用，不扩容、不释放
- 结构体必须固定大小，字段不可变
- 禁止使用会发生自动扩容的容器（如 List）

详细的内存管理策略已放入附录中

参数更新协议：避免“半写半读”

由于 Windows 与实时域之间通过共享内存通信，必须确保参数写入具有“帧一致性”。

推荐使用如下更新协议：

- Windows 先写入所有参数字段
- 最后写入一个控制字段（标记数据已写完）
- 实时域仅在检测到控制字段变化时才读取整帧参数
- 完成处理后由实时域写入状态字段供 Windows 查询

该机制可避免实时域读取到“未写完”的参数，确保数据始终一致。

（共享内存一致性细节参见第 3 章）

实时域的错误处理机制

Windows 程序常用的异常模式（throw Exception / MessageBox）在实时域不可使用。

NoTime 推荐：

- 实时域通过“错误码 + 状态机”报告故障
- Windows 负责显示报警、日志、错误详情
- 实时域进入安全态（SAFE STOP），等待上位机复位

这样做可以确保系统在故障情况下仍可预测、可恢复。

共享内存的一致性与结构对齐

Windows 和 Runtime 必须使用完全一致的共享结构体定义：

- 字段顺序一致
- 字段类型一致
- 禁止引用类型
- 禁止动态长度字段
- 建议使用 Snapshot 模式（每周期一次性读取）

结构体一致性是保证数据传输正确的基础。

两阶段验证流程（迁移的建议步骤）

迁移后的程序建议通过以下验证流程：

- 阶段 1：Windows 模式（run_in_ntf = false）
- runtime.exe 在 Windows 中运行
- 可以断点、调试、打印
- 验证逻辑、状态机、参数流程

阶段 2：实时模式（run_in_ntf = true）

- Runtime.exe 由 Windows 加载到 INtime
- 检查周期稳定性

- 监控执行时间是否超预算
- 验证是否存在阻塞、抖动、超时

这是确保迁移成功的关键步骤。

最终部署模型

迁移完成后的系统整体形态如下：

Windows.exe

- UI、配置管理、日志、通信
- 负责启动 Runtime
- 负责监控实时域的状态

runtime.exe（NoTime 实时域）

- 负责实时控制
- 周期执行
- 使用固定线程与固定内存布局
- 专注于动作、状态机与执行
- 共享内存
- 用于参数下发与状态回传
- 单写多读、固定结构、可预测

这是用户需要充分理解的部署结构。

关于ProU

ProU是行业领先的 x86 实时机器控制平台。自 2015 年创立以来，我们持续拓展应用场景，技术已在 3C、半导体、新能源等领域快速落地。优易控通过将 x86 高算力与通用 OS 深度整合，实现从计算到执行的全链路实时控制，为制造业构建面向 AI 的核心控制与数据底座。