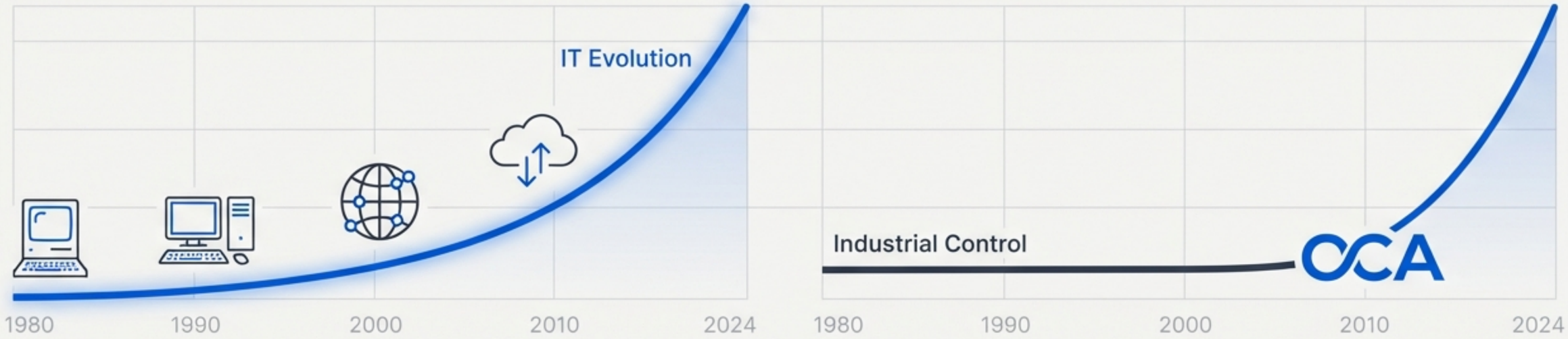
Closing the 40-Year Gap

How C++20 and commodity hardware are finally delivering the lost decades of IT progress to industrial control.

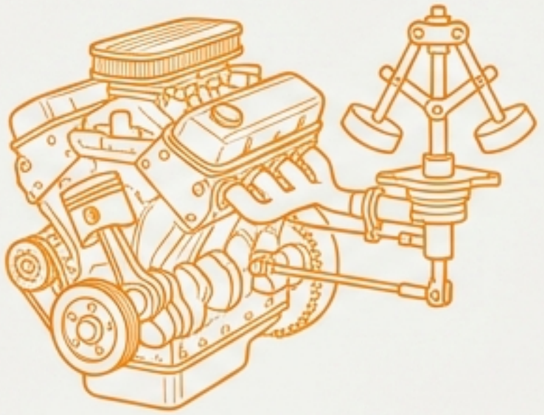


The IT world has evolved dramatically in 40 years. **Industrial** control has not. **It remains bound by the paradigms of the 1980s**—not because of technical limits, but because of business models. **Today, we break those chains.**

The Three Curses: Why Control Technology Stalled in the 1980s

Hardware Lock-in (The Iron Cage)

The dominant business model is not to sell the best controller, but to ensure the recurring sale of proprietary hardware. This is achieved by tying the programming language and compatible devices to a single vendor, stifling competition and innovation.



Suppressed Processor Potential (The Governor)

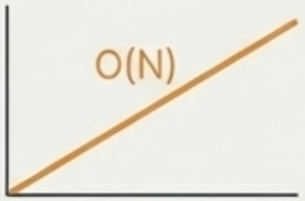
Modern processors in a $30 Raspberry Pi or a standard PC vastly outperform the specialized ASICs in traditional PLCs. Yet, legacy architectures fail to utilize even a fraction of this power, leaving immense computational potential on the table.

Software Technology Disconnection (The Time Capsule)

Control logic development is trapped in "Structured Programming," a paradigm from 40 years ago. Modern software engineering principles like true object-orientation (inheritance, polymorphism) are absent, leading to massive inefficiencies in development and maintenance.

This Isn’t “PLC-Style OOP.” This is a Fundamental Paradigm Shift.

Consider a required logic change across 100 similar machines (e.g., cylinder control).

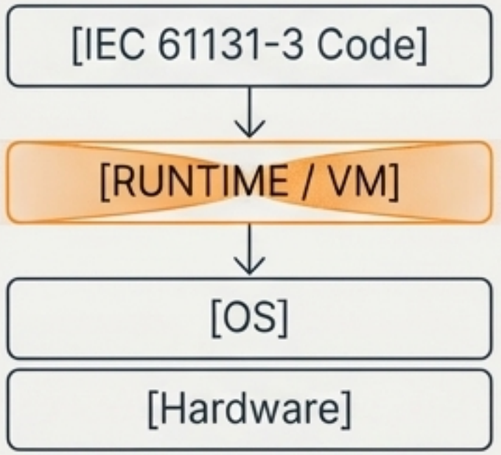
	Traditional PLC (Structured / “Function Block”)	OCA (True Object-Oriented)
Paradigm	**Copy & Paste.** Function Blocks are just reusable snippets. They cannot be inherited or extended.	**Inheritance & Polymorphism.** A single change to a parent class is automatically inherited by all child classes.
Scope of Change	Manually modify **100** separate program locations or FB instances.	Modify **1** parent class definition.
Effort & Risk	**100x the effort.** Every copied instance is a new potential point of failure that must be individually tested.	**1/100th the effort.** Test the single parent class to validate the change for all instances.
Scalability	**O(N) Complexity.** Cost and risk scale linearly with the number of machines. 	**O(1) Complexity.** Cost and risk are constant, regardless of the number of machines. 

PLC “OOP” is just encapsulation. OCA leverages C++20’s inheritance and polymorphism to structurally eliminate redundant work and reduce maintenance costs.

A PC as a PLC Emulator, or as the Ultimate Controller?

The Emulator Approach (Existing Software PLCs)

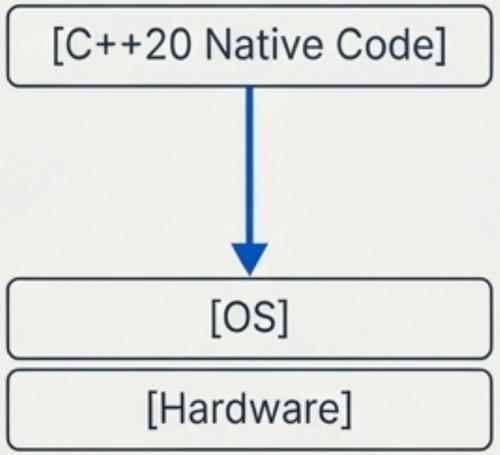
This approach uses a "Runtime" or Virtual Machine to execute legacy PLC languages (IEC 61131-3) on a PC. It prioritizes compatibility over performance.



The runtime layer introduces significant overhead and jitter, preventing the system from harnessing the processor's full native speed.

The Native Approach (OCA)

OCA is a C++20 native framework. There is no runtime. The control logic is compiled directly into an optimized binary that runs on the OS, unlocking maximum performance.

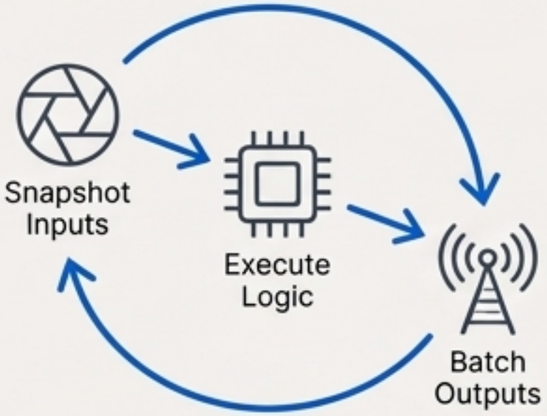


This architectural purity eliminates overhead, achieving cycle times of **3µs or less**—even on general-purpose operating systems like Windows and Linux.

Performance & Robustness: The End of the “Unstable PC” Myth

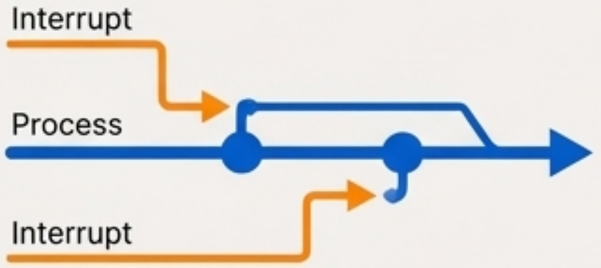
1. Microsecond Performance (3µs Cycle Time)

By eliminating architectural overhead, OCA converts 100% of the processor's power into control performance. This enables a level of precision that was previously impossible on general-purpose hardware, rivaling and exceeding dedicated systems.



2. Structural Data Integrity (The Process Image)

OCA enforces the core principle of PLC robustness. Every cycle follows a strict `Snapshot Inputs -> Execute Logic -> Batch Outputs` sequence. This structurally prevents data from changing mid-calculation due to I/O or other tasks, guaranteeing deterministic behavior.

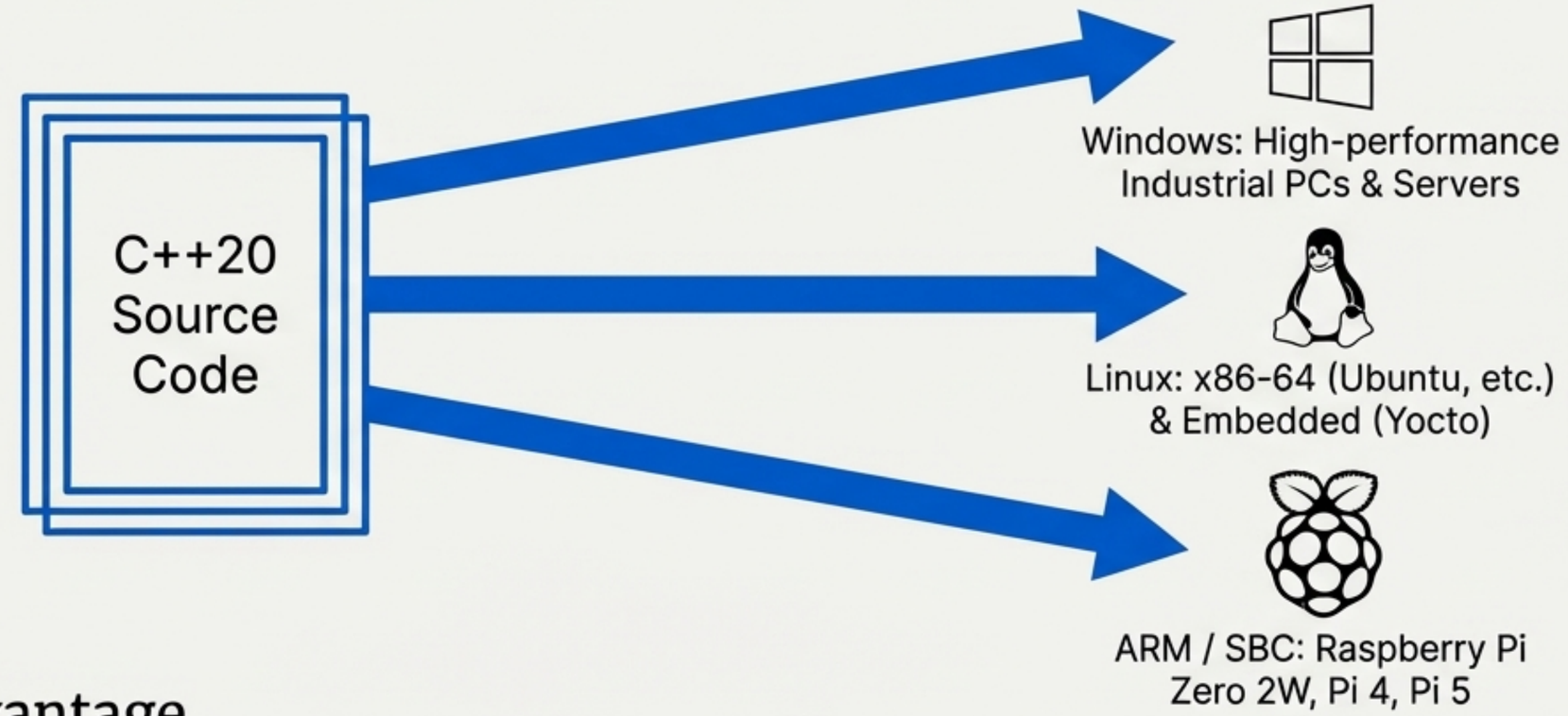


3. Race-Free Concurrency (Cooperative Interrupts)

High-priority tasks don't preemptively interrupt active logic. Instead, interrupts are handled safely at the boundary between logic blocks (“networks”). This mechanism prevents data corruption and race conditions by design, ensuring stability in a multi-threaded environment without developer intervention.

True Cross-Platform Freedom: Write Once, Run Anywhere

Your control logic should not be tied to a specific piece of silicon. OCA is built on standard C++20, making your software asset truly portable.

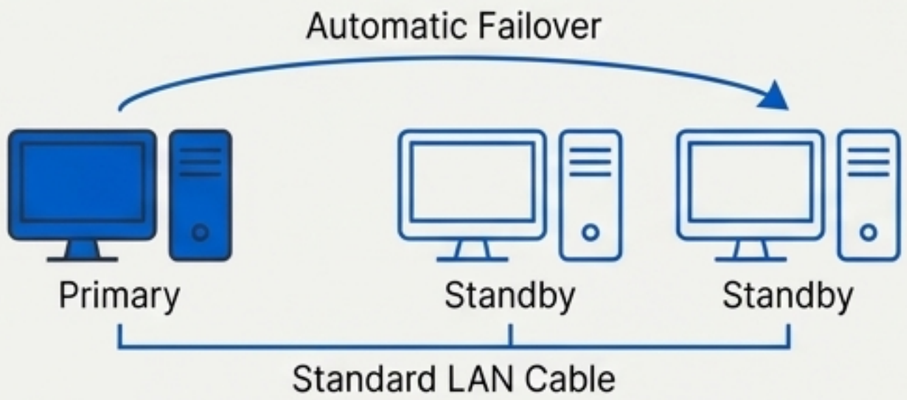


Strategic Advantage

Develop on a Windows PC, deploy on a low-cost Raspberry Pi. Instantly pivot hardware platforms during a chip shortage. OCA gives you the architectural freedom to make decisions based on cost, performance, and availability—not on vendor dictates. This is made possible by a design that strictly isolates and abstracts any OS-dependent code.

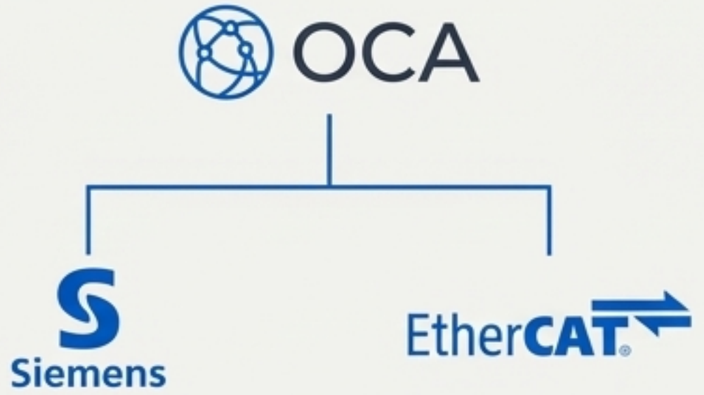
High-End Features, Democratized

1. N-Way Redundancy on Commodity Hardware



Achieve mission-critical high availability without expensive proprietary controllers or tracking cables. OCA's N-Way redundancy is a standard software feature. Simply connect multiple PCs or devices with a standard LAN cable to create a primary/standby cluster with automatic failover. Data synchronization is validated with CRC checks to ensure zero corruption.

2. Multi-Protocol Control in a Single Project



Break free from vendor-specific communication ecosystems. OCA's abstracted communication layer allows you to natively control devices from different manufacturers using different protocols within the same application.
Supported Protocols: Siemens S7, EtherCAT, and more.
Supported Protocols: Siemens S7, EtherCAT, and more.

The Framework Manages Complexity. You Write the Logic.

Implementation Steps

1. **Inherit**: Start by inheriting from the provided `NetworkBase` class.
2. **Implement**: Write your control logic inside the `OnExecute()` method.

```
class MyLogic : public FaFramework::NetworkBase
{
public:
    MyLogic(const std::string& id) : NetworkBase(id)
    {
        // Register your input and output tags here
        RegisterInput("Sensor_A");
        RegisterOutput("Actuator_B");
    }
protected:
    void OnExecute() override
    {
        // Your logic is pure and simple
        bool sensor_state = GetBool("Sensor_A");
        SetBool("Actuator_B", sensor_state);
    }
};
```

The Framework's Guarantee

When you write code this way, the OCA framework **automatically** handles:

- ✓ **Data Quality Checks**: `OnExecute()` will not run if any input tag has 'Bad' quality.
- ✓ **Thread-Safe I/O**: All `Get` and `Set` calls are inherently thread-safe. No manual locking required.
- ✓ **Exception Handling**: System-level exception handling prevents a single logic error from crashing the controller.

Deployment Flexibility

Choose your module strategy:

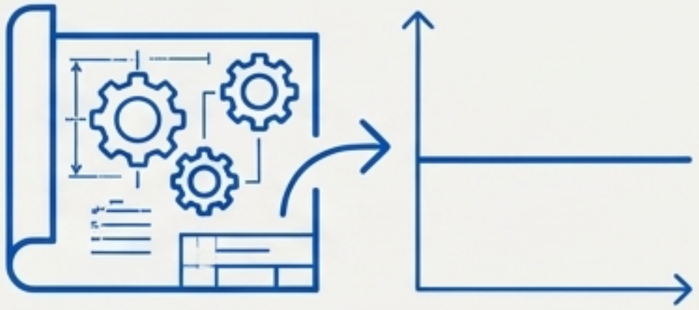
Static Linking: Compile logic into a single, secure executable.

Dynamic Linking (DLL/so): Load and update logic modules without stopping the main engine.

The 40-Year Gap is Closed.

1. Escape Vendor Lock-In Forever

Use any PC, IPC, or SBC that runs Windows or Linux. Free your business from hardware dependencies and supply chain volatility.



2. Achieve Radical Productivity & Maintainability

Leverage true object-oriented C++20 to slash the cost and risk of developing, scaling, and maintaining complex control software. Move from O(N) to O(1) complexity.

3. Future-Proof Your Systems by Default

With 3µs performance, built-in N-Way redundancy, and true cross-platform portability, you are building on an architecture designed for the next 40 years, not the last.

Object-Component Automation fully inherits the **safety and robustness** forged by decades of PLC development. It then injects the **speed, flexibility, and efficiency** of modern IT.

This transforms your **control system** from a defensive necessity into an offensive tool that accelerates your business.

The New Standard is Here.