What inspired me to create my own IDE platform for automation?

What made me start this journey? In short — frustration and curiosity.
I spent years working with automation, embedded systems, and low-level logic, and constantly ran into the same problem: simple ideas were buried under complexity. Either you had to rely on bloated proprietary PLC software, or dive into C-based firmware just to make a sensor-controlled blinking LED work. That might be acceptable for a final product — but it's terrible for prototyping and learning.

I wanted to create a tool where engineers — or even students — could describe logic modularly and visually, without losing control. Something like a software breadboard: plug in inputs, define states, set actions — done. No cloud dependency, no vendor lock-in, no steep learning curve.

Over time, this idea evolved into a logical IDE with an integrated soft-PLC, EFSM blocks, USB-based GPIO management, and even AI assistance for generating documentation, wiring diagrams, and logic templates.

To me, this is not about "replacing code" — it's about accelerating iteration. It's about allowing more people to experiment, build, learn, and bring their ideas to life.

Why an IDE should be more than just a code editor
Most environments for automation logic suffer from either overcomplication or excessive abstraction. On one end of the spectrum, you have traditional SCADA/PLC platforms that demand licenses, training, and strict hardware coupling. On the other, you see attempts to visualize logic a la Blockly — where the real behavior is detached from physical interfaces.

I aimed to break away from this binary model.

It wasn't just about "developing software," but about creating an environment for engineering thinking — where logic is designed as intuitively as wiring a circuit, and can be tested on hardware immediately.

Over time, the system evolved into a two-layer structure - **Research & Development Environment (R&D)**

Why a finite state machine?
A finite state machine is a structure with clearly defined states, transition conditions, and actions bound to the current state and triggering event.

What Does State Machine Logic Offer Instead?

Beeptoolkit formalizes logic as a finite state machine with a transition table.

This means:

- The system's behavior is structured and transparent
- All transitions are controlled through conditions and events
- Actions are tied to states, not code impulses
- Logic changes do not require recompilation — automata can be edited directly in the IDE

Instead of spending time re-implementing basic control logic, developers can focus on designing the scenario structure and defining system reactions to physical events.

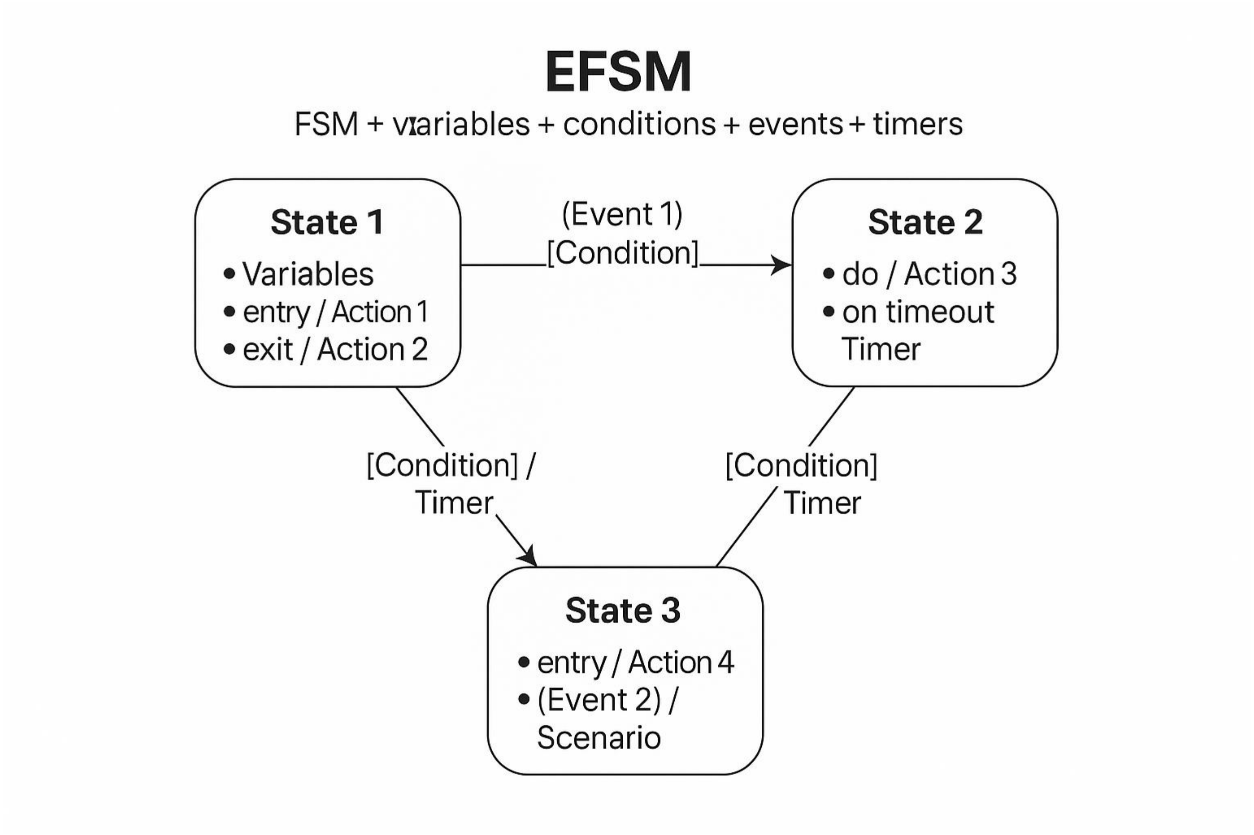
This brings several key advantages:

- System behavior becomes visual and formalized;

- There are no “floating” variables — each transition is predictable;
- Determinism makes debugging and verification much easier;
- It’s a natural way of thinking for engineers familiar with relay logic and transition diagrams.
- Logic design using finite state machines **(FSM)**

The core of the Beeptoolkit environment is based on automata programming — a method where system behavior is described not as a linear algorithm, but as a set of states and transitions between them triggered by events. This approach is especially valuable where behavior depends on conditions, flags, sensor states, and requires strict control over sequences.

IDE Software Logic Controller BEEPTOOLKIT - A scalable method that combines the strengths of the Moore and Mealy models while remaining fully formalizable.



How to Classify This Technically

Beeptoolkit is a **visual programming platform** based on **Extended Finite State Machines (EFSM)**:

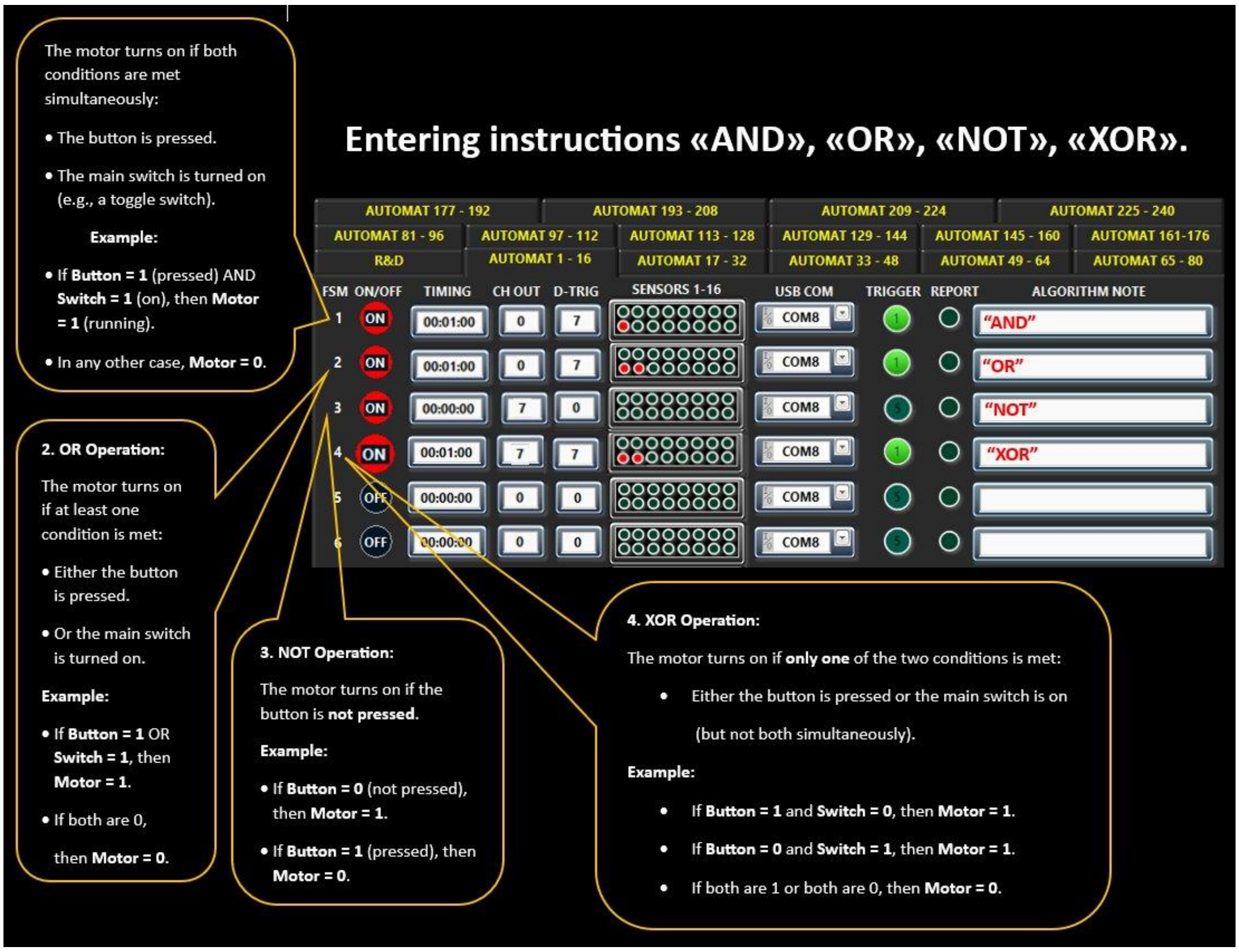
EFSM = FSM + variables + conditions + events + timers

This model is commonly used in **SCXML**, **UML Statecharts**, **modern PLC languages** (e.g. **SFC**, **ST**), and **low-code platforms**.

Therefore, Beeptoolkit employs an **EFSM** model where **Moore-type states** are used as **containers** for:

- **Event handling** (as in Mealy models)
- **Transition conditions** (as in rule-based logic)
- **Timers** (discrete time)
- **Event processing routines**

This approach is **scalable**, **highly visual**, and ideal for **prototyping** and **R&D**. It combines the strengths of both **Moore and Mealy** models while remaining **fully formalizable**.



Entering instructions «AND», «OR», «NOT», «XOR».

FSM	ON/OFF	TIMING	CH OUT	D-TRIG	SENSORS 1-16	USB COM	TRIGGER	REPORT	ALGORITHM NOTE
1	ON	00:01:00	0	7		COM8			"AND"
2	ON	00:01:00	0	7		COM8			"OR"
3	ON	00:00:00	7	0		COM8			"NOT"
4	ON	00:01:00	7	7		COM8			"XOR"
5	OFF	00:00:00	0	0		COM8			
6	OFF	00:00:00	0	0		COM8			

The motor turns on if both conditions are met simultaneously:

- The button is pressed.
- The main switch is turned on (e.g., a toggle switch).

Example:

- If **Button = 1** (pressed) AND **Switch = 1** (on), then **Motor = 1** (running).
- In any other case, **Motor = 0.**

2. OR Operation:

The motor turns on if at least one condition is met:

- Either the button is pressed.
- Or the main switch is turned on.

Example:

- If **Button = 1** OR **Switch = 1**, then **Motor = 1.**
- If both are 0, then **Motor = 0.**

3. NOT Operation:

The motor turns on if the button is **not pressed**.

Example:

- If **Button = 0** (not pressed), then **Motor = 1.**
- If **Button = 1** (pressed), then **Motor = 0.**

4. XOR Operation:

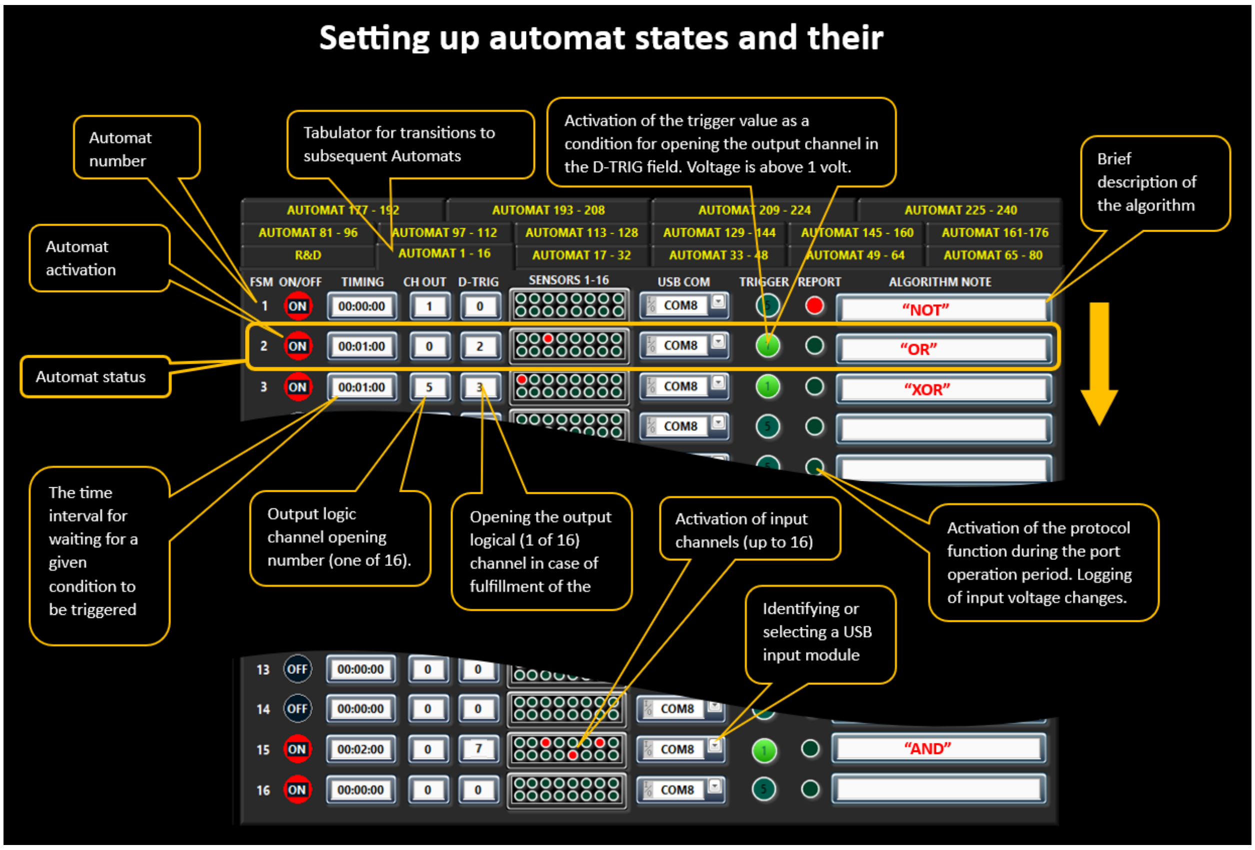
The motor turns on if **only one** of the two conditions is met:

- Either the button is pressed or the main switch is on (but not both simultaneously).

Example:

- If **Button = 1** and **Switch = 0**, then **Motor = 1.**
- If **Button = 0** and **Switch = 1,** then **Motor = 1.**
- If both are 1 or both are 0, then **Motor = 0.**

How is automata-based programming implemented in Beeptoolkit?



Setting up automat states and their

FSM	ON/OFF	TIMING	CH OUT	D-TRIG	SENSORS 1-16	USB COM	TRIGGER	REPORT	ALGORITHM NOTE
1	ON	00:00:00	1	0		COM8			"NOT"
2	ON	00:01:00	0	2		COM8			"OR"
3	ON	00:01:00	5	3		COM8			"XOR"
						COM8			
13	OFF	00:00:00	0	0					
14	OFF	00:00:00	0	0		COM8			
15	ON	00:02:00	0	7		COM8			"AND"
16	ON	00:00:00	0	0		COM8			

Automat number

Automat activation

Automat status

Tabulator for transitions to subsequent Automats

Activation of the trigger value as a condition for opening the output channel in the D-TRIG field. Voltage is above 1 volt.

Brief description of the algorithm

The time interval for waiting for a given condition to be triggered

Output logic channel opening number (one of 16).

Opening the output logical (1 of 16) channel in case of fulfillment of the

Activation of input channels (up to 16)

Identifying or selecting a USB input module

Activation of the protocol function during the port operation period. Logging of input voltage changes.

Visual Logic as a Formula of Interaction

Unlike traditional *if-else* programming, logic in Beeptoolkit is structured as a **transition table** or a **state graph**.

This makes system behavior:

- **Visual** – suitable for validation and debugging
- **Predictable** – each transition can be explicitly traced
- **Reusable** – an automaton can be exported or adapted for other tasks

This structured approach is particularly valuable in research and educational environments, where it is essential to capture, reproduce, and build complex behaviors from simple and understandable components.

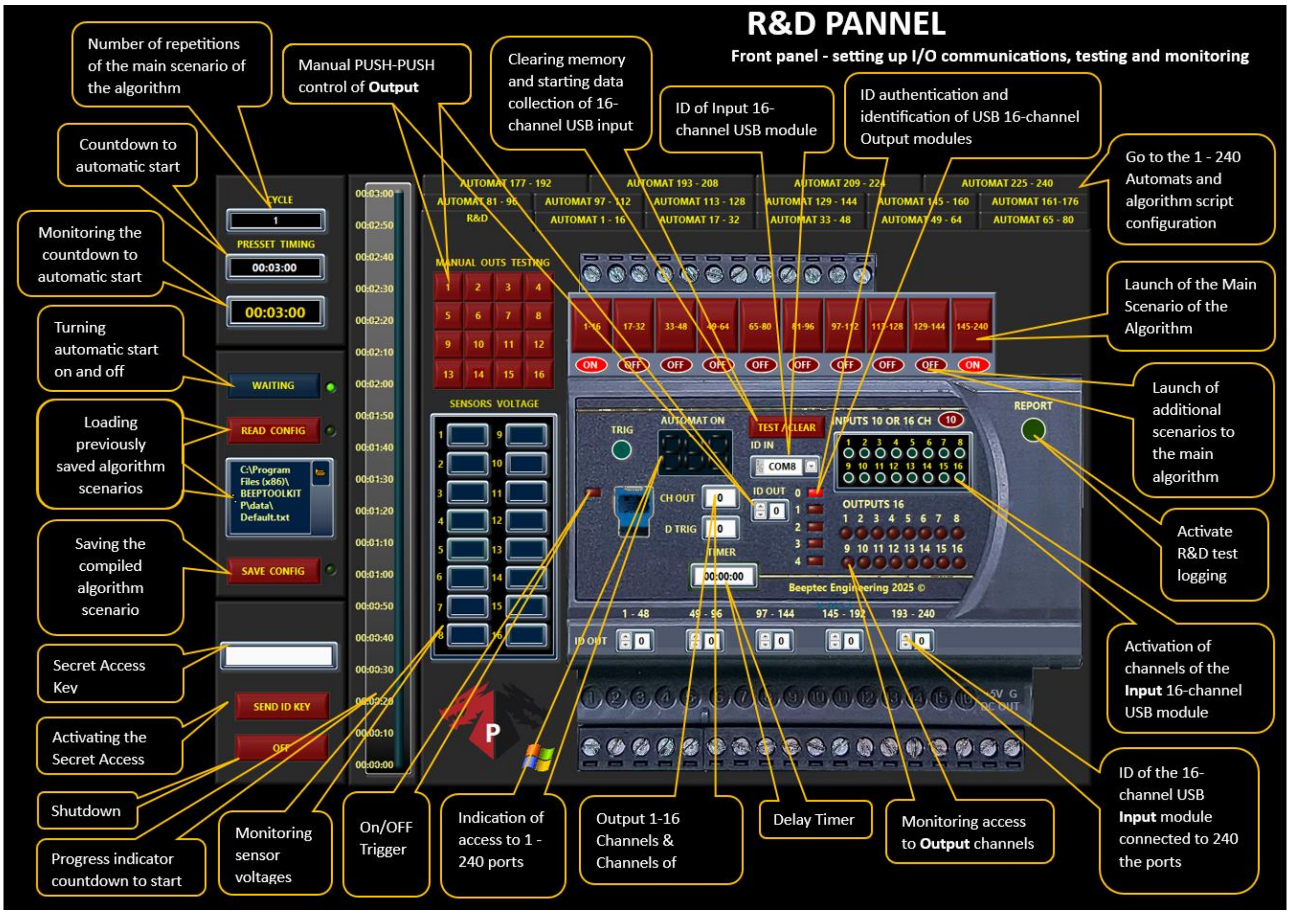
Behavior Configuration:

- Event-driven logic with support for timers, flags, and external conditions
- Logic generation from templates or natural-language commands

---

Execution Layer (Runtime PLC)

- Interfaces directly with **GPIO** (DAC/ADC) modules over **USB, UART, RS485**
- Executes state machine logic **independently of cloud connectivity** Scales from **a single module** to **a distributed multi-module system**


R&D PANNEL
Front panel - setting up I/O communications, testing and monitoring
Number of repetitions of the main scenario of the algorithm
Manual PUSH-PUSH control of Output
Clearing memory and starting data collection of 16-channel USB input
ID of Input 16-channel USB module
ID authentication and identification of USB 16-channel Output modules
Go to the 1 - 240 Automats and algorithm script configuration
Countdown to automatic start
Monitoring the countdown to automatic start
Turning automatic start on and off
Loading previously saved algorithm scenarios
Saving the compiled algorithm scenario
Secret Access Key
Activating the Secret Access
Shutdown
Progress indicator countdown to start
Monitoring sensor voltages
On/OFF Trigger
Indication of access to 1 - 240 ports
Output 1-16 Channels & Channels of
Delay Timer
Monitoring access to Output channels
ID of the 16-channel USB Input module connected to 240 the ports
Activation of channels of the Input 16-channel USB module
Activate R&D test logging
Launch of additional scenarios to the main algorithm
Launch of the Main Scenario of the Algorithm
CYCLE
PRESSET TIMING
00:03:00
WAITING
READ CONFIG
C:\Program Files (x86)\BEEPTOOLKIT P\data\ Default.txt
SAVE CONFIG
SEND ID KEY
OFF
MANUAL OUTS TESTING
SENSORS VOLTAGE
AUTOMAT 177 - 192
AUTOMAT 193 - 208
AUTOMAT 209 - 224
AUTOMAT 225 - 240
AUTOMAT 81 - 96
AUTOMAT 97 - 112
AUTOMAT 113 - 128
AUTOMAT 129 - 144
AUTOMAT 145 - 160
AUTOMAT 161-176
R&D
AUTOMAT 1 - 16
AUTOMAT 17 - 32
AUTOMAT 33 - 48
AUTOMAT 49 - 64
AUTOMAT 65 - 80
1-16
17-32
33-48
49-64
65-80
81-96
97-112
113-128
129-144
145-240
ON
OFF
TRIG
AUTOMAT ON
TEST / CLEAR
INPUTS 10 OR 16 CH
ID IN
COM8
REPORT
CH OUT
D TRIG
ID OUT
TIMER
00:00:00
OUTPUTS 16
Beeptec Engineering 2025 ©
ID OUT
1 - 48
49 - 96
97 - 144
145 - 192
193 - 240

Why Not Use Existing Solutions?

A Comparison Between the Beeptoolkit IDE Approach and Traditional Automation Tools, Including Their Limitations and Learning Curve

This question comes up frequently — and for good reason. The automation market already offers a variety of established tools: from classical PLCs to modern visual development environments. However, most of them were either designed for different purposes or impose limitations that make them unsuitable for rapid and open prototyping.

Traditional PLC Systems

Platforms such as Siemens, Schneider, Mitsubishi, and others are excellent for industrial automation tasks. They are reliable, certified, and scalable. However, they come with several constraints:

- **Hardware Lock-in**: Tied to specific hardware and often reliant on proprietary communication protocols
- **License-Restricted Development**: Development requires expensive proprietary software licenses
- **Legacy Development Environments**: Non-standardized IDEs with outdated user interfaces
- **Steep Learning Curve**: Require knowledge of domain-specific languages (e.g., ST, LD, FBD)
- **Poor Suitability for Prototyping**: Not designed for rapid testing or iterative development cycles

When the goal is research, debugging, or education, this approach becomes too time-consuming and cognitively demanding.
Many engineers and developers, especially those experienced in embedded systems, prefer to write control logic manually using C, C++, Python, MicroPython, the Arduino stack, or bare-metal code. This approach seems universal: you retain full control over behavior, use familiar tools, and avoid dependency on external environments.

But this control often comes at the cost of resource intensity, fragmentation, and excessive complexity — especially in tasks requiring modularity, quick hypothesis testing, or collaboration among multiple specialists.

Repeating the Same Solutions Manually

Almost every project involving manual programming ends up reimplementing the same standard patterns:

- Button debouncing
- Timers and delays (millis() in Arduino, HAL_Delay() in STM32)
- Tracking current states
- Transition logic (if/switch-case)
- Error handling and restart routines

This process is not only tedious but also discourages reuse — the code is typically tailored to a specific task, on a specific architecture, tightly coupled with hardware.

Lack of Logic Formalization

In manual development, there is often no clear separation between state, condition, and action, leading to:

- Chaotic growth of variables
- Transition logic bugs
- Maintenance difficulties — especially when adding new operating modes

As a result, even small behavior changes may require significant rewrites of the codebase.

Problems with Scalability and Modularity

Manual programming is convenient at the beginning. But when automation extends beyond a single motor or sensor, challenges arise:

- Different modules in the system may conflict or duplicate functionality
- Synchronizing states across multiple devices becomes complex
- Testing and verifying system behavior becomes harder
- Adding logging or telemetry requires touching every part of the logic

---

Arduino-Based Systems

Arduino platforms are widely used in hobbyist and educational environments and are often the first choice for beginners due to their simplicity and affordability. However, they introduce several limitations in a professional automation context:

- **Low-Level Programming Required**: Requires C/C++ code development, which limits accessibility for non-programmers
- **Limited State Management**: Lacks built-in support for structured state machines or automation workflows
- **No Native Logic Visualization**: All behavior must be coded manually — no native visual representation of automation logic
- **Not Scalable**: Designed for small-scale projects, not suitable for large I/O systems or distributed logic control
- **Unstructured Prototyping**: Rapid prototyping is possible but often lacks documentation, versioning, and maintainability

---

STM32 and Bare-Metal Embedded Systems

Microcontrollers like STM32 (from STMicroelectronics) provide high-performance and are commonly used in industrial and embedded systems. They offer excellent flexibility and control, but come with steep development overhead:

- **Complex Toolchain**: Requires integration of IDEs (e.g., STM32CubeIDE), compilers, debuggers, and firmware libraries
- **High Entry Barrier**: Demands deep knowledge of hardware-level programming, registers, memory management
- **Time-Consuming Development**: Prototyping takes significantly longer due to manual setup of peripherals, interrupts, and logic

- **No Built-In State Machine Logic**: EFSMs must be implemented from scratch, increasing complexity and risk of bugs
- **Limited Modularity for Logic**: Reusability and modular design require significant upfront architectural planning

---

ESP32, Raspberry Pi, and Similar Boards

Boards like ESP32 and Raspberry Pi offer Wi-Fi/Bluetooth connectivity and general-purpose computing, making them popular for IoT and robotics. However, they come with mixed pros and cons for logic automation:

- **ESP32 (Microcontroller)**:
    - Suitable for lightweight embedded automation
    - Still requires C/C++ or MicroPython programming
    - Limited support for structured visual automation logic
    - IDEs (like PlatformIO, Arduino IDE) are not tailored for state-based automation
- **Raspberry Pi (Single-Board Computer)**:
    - Powerful for multimedia, AI, and high-level control logic
    - Relies on general-purpose OS (Linux), leading to unpredictable timing behavior
    - Not designed for deterministic real-time control
    - Overhead of maintaining OS updates and dependencies

Beeptoolkit - IDE Soft Logic Controller bridges the gap between professional automation needs and rapid prototyping by offering:

- **Visual Finite State Machine (EFSM) Logic Design** — no need to write C/C++ code
- **Hardware Integration via USB (DAC/ADC)** — plug-and-play with up to 1600 I/O channels
- **No Vendor Lock-In** — compatible with any x86 PC architecture
- **No Real-Time OS Dependencies** — self-contained runtime for predictable behavior
- **Open Logic Architecture** — export, reuse, and scale logic blocks across projects
- **Beginner-Friendly, Expert-Ready** — ideal for both fast concept validation and complex deployments

---

Platform and Compilation Dependency

Each code modification requires:

- Recompilation
- Flashing the microcontroller
- System restart

This significantly slows down iteration speed, especially during the research phase when the system's behavior must be adjusted to real-world conditions and evolving hypotheses.

Collaboration Challenges

A manually written project is harder to:

- Explain to other team members
- Document in an engineering-friendly format
- Verify against specifications

It may be perfectly clear to its original author but is often opaque and difficult to understand for others — which is critical in R&D or educational environments.

Conclusion

Manual programming remains essential for low-level optimization, protocol handling, and hardware interfacing. However, when it comes to behavior logic, prototyping, education, and reuse, it:

- Slows down the development process
- Requires a high level of expertise
- Fails to offer structured behavior description or easy validation

Beeptoolkit addresses these pain points, making automation logic easier to build, share, and evolve.

Here is the English translation of your text:

Why Was Beeptoolkit Designed as a PC-Based Environment Rather Than Embedded in a Microcontroller?

Most industrial and hobbyist automation systems today are based on embedding logic directly into a microcontroller. This approach seems logical: placing the code "closer to the hardware" reduces latency, simplifies the architecture, and removes the dependency on an external host.

However, Beeptoolkit deliberately takes a different path: the logic environment runs on a PC, while microcontrollers and external modules are treated as low-level executors — handling I/O interfaces, ADCs, PWM, and similar functions.

This decision is not about implementation convenience, but rather an engineering strategy aimed at flexibility, scalability, and faster iteration cycles.

Beeptoolkit Implements an Architecture Where:

- The **PC acts as the brain**: it holds and executes the logic, makes decisions, manages state history, and controls state transitions.
- The **MCUs** (STM32, USB I/O modules, GPIO controllers, etc.) act as **sensors and actuators**, handling analog and digital signal operations.

This Separation Enables:

- Modifying and debugging logic **without recompiling or reflashing** the microcontroller

- Transferring behavior across projects **without firmware changes**
- Maintaining a **single executable logic context**, suitable for analysis, logging, and simulation

---

Accelerating Iterations and Lowering the Entry Barrier

When logic is executed on the PC, developers can:

- Make real-time modifications
- Edit and test without even connecting hardware
- Quickly switch between modes, scripts, and scenarios

This dramatically speeds up prototyping — especially valuable in lab development, hypothesis testing, and educational environments.

Eliminating Re-Flashing and Decoupling from Firmware Stacks

Each attempt to implement logic on a microcontroller (MCU) requires:

- Creating a custom event loop
- Accounting for hardware limitations
- Integrating support for inputs, timers, logic, and delayed events
- Testing and flashing, often with a full system reboot

This becomes even more complicated when working with multiple controllers, non-standard protocols, or unstable communication links.

Beeptoolkit, in contrast, works with standardized data streams (UART, USB CDC, HID, SPI over USB), where all firmware on the controller acts as a **state gateway** only.

❗ **Crucially:** Implementing logic outside the MCU **does not** mean sacrificing real-time capability — provided a correct architecture is used for handling inputs and outputs (see the next section on determinism and latency).

Hardware-Agnostic Flexibility

Beeptoolkit was designed from the ground up as a platform-independent environment. It is not tied to a specific microcontroller or hardware vendor. Logic execution is performed on a PC, while interaction with physical devices happens via **universal interfaces** like USB, UART, and RS-485.

This provides the user with **freedom to choose hardware modules** based on project needs and local market availability.

Support for Widely Available Peripheral Modules

Beeptoolkit works with the types of load drivers, signal converters, and I/O modules that are widely used in engineering and are easily sourced from platforms like **AliExpress**, **Amazon**, **eBay**, or **local distributors**. This hardware does **not require firmware development** and connects easily via general-purpose interfaces.

Beeptoolkit's Dual-Core Architecture:

- **Logic Core**: Runs on a PC and ensures modular behavior, state transition logic, and validation. It includes the IDE and the soft logic controller functioning as a unified system.

This approach enables:

- A sharp **reduction in time** needed to build functional prototypes
- Simulation of system behavior **without any hardware**
- Easy **logic migration between PCs** using project file export/import with documented state transition tables and logic scripts

Implementing Beeptoolkit as a **PC-based logic environment** is not a compromise — it is a principle. It represents a **deliberate rejection of firmware routines** and MCU limitations at the logic development and verification stage.

What Logical Instructions Are Available in Beeptoolkit and How Are They Related to Binary Logic?

Beeptoolkit was designed based on industrial control logic principles — similar to those found in classic PLC systems:

- Tight binding of logic to **physical inputs and outputs**
- Operation in **discrete time cycles**
- Behavior determined by the **current system state**

This foundation enabled the implementation of a **precise and sufficient set of instructions**, rooted in binary logic but capable of expressing complex automation scenarios.

Вот перевод вашего текста на английский язык:

What is Binary Logic in the Context of Automation?

Binary logic refers to working with **discrete signals** that can exist in one of two states — **ON/OFF**, **1/0**, or **HIGH/LOW**. It is the fundamental principle behind all control systems in automation:

- A button is either pressed or not
- A sensor is triggered or not
- An output is active or not

This logic maps directly to physical interfaces such as **GPIOs**, **relays**, **transistor switches**, and **optocouplers**, making it a universal language for connecting digital logic with the real world.

Classes of Logical Instructions in Beeptoolkit

In Beeptoolkit, logical instructions are **attached to states** within a state machine and are executed **once upon entry** or **periodically while the state is held**. The main instruction categories include:

📌 **Transition Conditions Between States:**

- If input X = 1
- If timer Y has expired
- If input A = 1 AND input B = 0
- If logical flag Z = true

These are **logical expressions** comparing inputs, flags, and internal variables.

📌 **Actions on Entering a State:**

- Set output X to 1
- Reset output Y
- Start a timer
- Save the value of input A to an internal variable

These are **deterministic control instructions** for physical or logical outputs.

📌 **Support for Flags, Variables, and Counters:**

- Binary flags (true/false)
- Trigger counters
- Buffers for last known values

In Beeptoolkit, this logic is implemented **entirely within the state machine**, using conditional checks and timing — **without interrupts**, but with **event- and logic-driven transitions**.

Conclusion

Beeptoolkit is evolving as a **universal, engineering-oriented automation environment** that doesn't impose a rigid architecture but **introduces a structured mindset**. It's not just a visual editor, and it's not just a PLC emulator — it’s a **new way of thinking** about logic, modularity, and system control.

Much technical and experimental work still lies ahead, but the direction is clear: to make automation **accessible**, **intuitive**, and **extensible** for those who **design**, **teach**, **research**, and **implement**.

Beeptec Engineering 2025
Autor Alexander Kapitulsky – CEO
E-Mail: info@beeptoolkit.com
Site https://beeptoolkit.com/
YouTobe: https://www.youtube.com/@beeptecengineering9884

BEEPTOOLKIT