Ouroboros: The Case for Optical Feedback in Autonomous Embedded Agents

Abstract

The trajectory of artificial intelligence in software development has largely been defined by a decoupling of intent from reality. We have built "blind" giants—massive reasoning engines housed in data centers, generating code based on statistical probabilities and verifying it against abstract compiler outputs. In the pristine environment of pure software, this is sufficient. A unit test passes or fails; a syntax error is caught or it is not. However, in the realm of embedded systems—where code must interact with the chaotic physics of silicon, voltage, and light—this blindness is a fatal flaw.

Quantara presents "Ouroboros," an architectural doctrine and technical framework designed to close the loop. We argue that true autonomy in embedded engineering requires the agent to *see* the physical manifestation of its code. By synthesizing the low-cost ubiquity of the ESP32-2432S028 "Cheap Yellow Display" (CYD), the standardized bridging capability of the Model Context Protocol (MCP), and the emergent visual reasoning of Multimodal Large Language Models (MLLMs), we propose a system where the machine builds, observes, and corrects in a continuous cycle of reinforcement.

Crucially, we address the fragility of autonomous hardware interaction through the "Enforcer"—a secondary, hardwired microcontroller acting as a physical watchdog. This report details the engineering reality of constructing such a system, challenging the exclusionary industrial standards of Hardware-in-the-Loop (HIL) testing and democratizing the capability for verified, self-correcting device fabrication. This is not merely a method for automating code; it is a blueprint for machines that possess the agency to recover from their own failures.

---

1. Introduction: The Blind Giant and the Democratization Gap

1.1 The Disconnect of the Disembodied Mind

The current state of AI-assisted coding is akin to a master architect who is locked in a dark room, capable of drafting perfect blueprints but forbidden from ever visiting the construction site. Large Language Models (LLMs) like GPT-4 and Claude 3.5 Sonnet are proficient at generating C++ syntax. They understand the semantics of the Arduino framework. They can write a valid tft.drawCircle(120, 160, 50, ILI9341_RED); command without hesitation.

However, the model has no concept of whether the screen is actually on. It does not know if the backlight pin (GPIO 21 on the CYD) was initialized high or if the specific batch of LCD panels used in production has a shifted color gamut that renders "red" as a washed-out pink. It cannot see if the coordinate (120, 160) falls under a physical crack in the glass or if the SPI bus speed is too high, causing digital noise artifacts (Moiré patterns) to obscure the graphic.

When an agent operates blindly, relying solely on the successful exit code of a compiler (exit status 0), it is operating in a hallucinated reality. It assumes that a valid binary equates to a functional device. Any embedded engineer knows this is a lie. A binary that compiles perfectly can still brick a device, lock up the USB stack, or produce a blank white screen. In the Quantara philosophy, a coding agent that cannot see its output is not an engineer; it is a random number generator with a compiler attached.

1.2 The Industrial Walled Garden: HIL and Verification

The industry has solved this problem, but the solution is locked behind a paywall so high it effectively excludes 99% of the world's innovators. This is the domain of Hardware-in-the-Loop (HIL) testing.

In the automotive and aerospace sectors, companies like National Instruments (NI), dSPACE, and Vector rule supreme. They build "rigs"—massive racks of proprietary hardware designed to simulate the physical world for an Electronic Control Unit (ECU). A dSPACE MicroAutoBox, a standard tool for testing automotive code, can cost upwards of $15,000 for the base unit alone. The software licenses to run these rigs are equally exorbitant, often requiring annual subscriptions in the thousands of dollars.

These systems are miracles of engineering. They offer microsecond-level latency, FPGA-based signal injection, and certification-grade reliability. But they represent the "Old World" of centralized, capital-intensive development. They are designed for Tier 1 suppliers creating braking systems for millions of cars, not for the boutique hacker building a privacy-centric mesh communicator or a decentralized sensor node.

This creates a "Democratization Gap." The independent developer is forced to rely on "Software-in-the-Loop" (SIL) simulation (like Wokwi or QEMU), which models the logic but strips away the physics. SIL cannot simulate the voltage drop when a WiFi radio transmits a packet. It cannot simulate the specific visual artifacting of a cheap TFT driver. It sanitizes the chaos of the real world, leaving the developer—and their AI agent—unprepared for reality.

1.3 The Quantara Ethos: Harm Reduction via Autonomy

Quantara approaches this problem from a specific philosophical angle: "Harm Reduction." In the context of technology, harm reduction means mitigating the risks of deploying insecure, untested, or fragile hardware into the wild.

We are entering an era where billions of IoT devices will be deployed, many of them running code generated by AI. If this code is verified only against a simulation, we risk flooding the physical world with "zombie" devices—gadgets that function in the lab but fail in the field, creating security vulnerabilities (e.g., a smart lock that fails open when the battery droops) or environmental waste (bricked devices filling landfills).

Ouroboros is our answer. By building a local, optical, self-correcting loop, we empower the independent engineer to achieve industrial-grade verification without the industrial price tag. We use the Cheap Yellow Display not because it is the best hardware, but because it is the *hardest* hardware—cheap, undocumented, and inconsistent. If an agent can autonomously build, verify, and repair a UI on a CYD, it can build anything.

---

2. The Architecture of Ouroboros

The Ouroboros system is defined by a triad of control elements—Subject, Observer, and Agent—stabilized by a fourth element of physical enforcement. This architecture is designed to be resilient to the three inevitabilities of embedded development: software crashes, hardware glitches, and communication deadlocks.

2.1 The Subject: The Cheap Yellow Display (CYD)

The "Subject" is the target of our creation. We have selected the **ESP32-2432S028**, known universally in the maker community as the "Cheap Yellow Display" (CYD).

2.1.1 Anatomy of a "Hostile" Subject

The CYD is a perfect specimen for this research because it represents the "supply chain reality" of modern electronics. It is available for approximately $15 USD, integrates a powerful microcontroller with a display, but is riddled with design compromises that an autonomous agent must navigate.

- **The Core:** The board is powered by an ESP32-WROOM-32 or, in some cost-reduced variants, an ESP32-PICO-V3. It features a dual-core XTENSA LX6 microprocessor running at 240 MHz. This provides sufficient compute for local logic but introduces multi-core concurrency challenges (e.g., race conditions between the UI task and the network task).
- **The Display:** The screen is a 2.8-inch TFT LCD with a resolution of 320x240 pixels. It typically uses an ILI9341 driver connected via SPI. However, the "supply chain lottery" means some boards arrive with ST7789 drivers, which have different initialization sequences and color formats (BGR vs RGB). An autonomous agent must be able to visually identify if the colors are inverted (blue dogs, yellow skies) and rewrite the driver initialization code accordingly.

- **The Touch Interface:** A resistive touch panel driven by an XPT2046 controller shares the SPI bus. Resistive touch is noisy and requires calibration. The agent must be able to guide a calibration routine—drawing targets, reading touch inputs, and calculating the transformation matrix.

2.1.2 The I/O Minefield

The pinout of the CYD is a constraint optimization puzzle. The ESP32 has a limited number of GPIOs, and the CYD uses almost all of them for onboard peripherals.

- **The Backlight Trap:** GPIO 21 controls the LCD backlight. If the agent treats GPIO 21 as a generic I/O or an I2C data line (SDA), the screen will go dark. This is a classic "silent failure" in blind development. The compiler reports success, the upload succeeds, but the device appears dead. Only an optical observer can detect this state.
- **The Extended I/O:** The board exposes a few "spare" pins on JST connectors (P3, CN1). These include GPIO 22, 27, and 35. However, GPIO 35 is input-only (no internal pull-up). An agent attempting to drive an LED on GPIO 35 will fail. The Ouroboros system must possess the datasheets in its context window or learn this limitation through trial and failure.

2.2 The Observer: Closing the Optical Loop

The "Observer" bridges the gap between digital intent and analog reality. It consists of a standard high-definition webcam (Logitech C920 or similar) mounted rigidly on a 2020 aluminum extrusion frame, looking directly down at the Subject.

2.2.1 From Pixels to Semantics

The raw feed from the camera is meaningless to a compiler. We must transmute pixel data into semantic understanding. This is achieved through the "Vision" capabilities of Multimodal Large Language Models (MLLMs), specifically **Claude 3.5 Sonnet** and **GPT-4o**.

- **Claude 3.5 Sonnet** is currently the gold standard for this application due to its superior spatial reasoning and ability to handle "needle in a haystack" visual queries. It can look at a 320x240 screen and accurately judge:
    - **Centering:** "Is the submit button equidistantly spaced from the left and right bezels?"
    - **Legibility:** "Is the font size sufficient for the contrast ratio against the background?"
    - **Artifacting:** "Are there horizontal tearing lines suggesting an incorrect SPI clock frequency?"

2.2.2 Visual Reinforcement Learning (ViRL)

The Observer enables a simplified form of Visual Reinforcement Learning.

1. **State:** The Agent holds a "Target State" in its context (e.g., "A red warning triangle in the center of the screen").

2. **Action:** The Agent compiles and uploads code.
3. **Observation:** The webcam captures the result.
4. **Critique:** The MLLM compares the capture to the intent. "The triangle is present, but it is rendered in orange, and it is offset 20 pixels to the right."
5. **Refinement:** The Agent calculates a correction vector (shift X coordinate -20, adjust RGB hex value) and iterates.

This loop replaces the human developer's eyes. It allows the system to perform "Visual Regression Testing" not on a DOM, but on physical photons.

2.3 The Agent: The Orchestration Layer

The "Agent" is the cognitive engine of Ouroboros. It is not embedded on the ESP32; it runs on a host PC (the "Controller"). It is a software entity that interacts with the world through the **Model Context Protocol (MCP)**.

The Agent's lifecycle is circular:

1. **Context Loading:** It ingests the project goals, the hardware schematics (CYD pinout), and the history of previous attempts.
2. **Code Generation:** It utilizes its LLM backend to write C++/Arduino code. It is instructed to write "defensively," adding serial logging checkpoints that help it diagnose boot failures.
3. **Tool Invocation:** It does not simply "output" code. It calls specific tools provided by the MCP server: write_file, compile, upload.
4. **Analysis:** It reads the serial monitor output (for boot logs) and the Observer's visual analysis (for UI verification).

2.4 The Enforcer: The Physical Watchdog

The most critical innovation in Ouroboros is the "Enforcer." In autonomous development, a "deadlock" is the enemy.
The ESP32's USB stack (CDC) is software-dependent. If the Agent writes code that crashes the CPU immediately upon boot, or puts the device into a deep sleep that powers down the USB PHY, the USB serial port will vanish from the host operating system.
A standard "blind" agent is helpless here. It cannot upload new code to fix the crash because the port required to upload the code is gone. The system is bricked until a human walks over and presses a button.
The Enforcer eliminates the human.

- **Hardware:** A secondary, robust microcontroller (Raspberry Pi Pico, Arduino Uno, or an FTDI bit-bang adapter) is physically hardwired to the CYD's **EN (Reset)** and **BOOT (GPIO 0)** pins.
- **Sovereignty:** The Enforcer operates outside the Subject's domain. It does not care if the ESP32 is crashed, sleeping, or on fire. When commanded, it executes physics.

- **The Rescue Sequence:**
  1. Agent detects "Port Missing" or "Upload Timeout."
  2. Agent invokes MCP tool: enforcer_rescue().
  3. Enforcer pulls **BOOT** (GPIO 0) LOW.
  4. Enforcer pulses **RESET** (EN) LOW for 200ms.
  5. Enforcer waits 100ms, then releases **BOOT**.
  6. The ESP32 wakes up in **ROM Bootloader Mode**. This mode is hardcoded in the silicon; it cannot be corrupted by user software.
  7. The USB port re-enumerates.
  8. The Agent uploads a "clean" firmware to restore functionality.

---

3. The Industrial Context: Why This Matters

To understand the necessity of Ouroboros, one must analyze the failures of the current embedded validation ecosystem. We are relying on methodologies that were designed for a different era of computing.

3.1 The Failure of Software-in-the-Loop (SIL)

"Software-in-the-Loop" simulation is the standard for low-cost testing. Tools like QEMU or Wokwi emulate the instruction set of the microcontroller. They are useful for logic verification (e.g., verifying that a JSON parser works), but they are deceptive for system verification.

The Reality Gap:

- **Electrical Noise:** A simulator provides a perfect 3.3V rail. The real CYD has a noisy LDO regulator that might droop when the WiFi radio transmits at 20dBm, causing the ADC (Analog-to-Digital Converter) readings to jitter. An agent calibrating a touch screen on a simulator will fail to account for this noise floor.
- **Timing Latency:** In a simulator, SPI transactions happen instantly or at a perfect clock divisor. In reality, capacitance on the PCB traces can round off the square waves, leading to data corruption at high speeds (e.g., 40MHz). A "sighted" agent can see the resulting visual corruption; a simulator cannot.
- **Peripheral Quirks:** The ESP32's hardware I2C implementation has historical bugs (glitches on the bus). Simulators often implement an "idealized" I2C peripheral that does not exhibit these silicon bugs.

SIL breeds a false sense of security. Ouroboros forces the agent to confront the messy reality of the hardware immediately.

3.2 The Stagnation of Visual Regression Testing (VRT)

Visual Regression Testing is a mature field in web development. Tools like **Applitools Eyes** and **Percy** allow developers to automatically check if a CSS change broke the layout of a website.

However, these tools are fundamentally incompatible with embedded development.

- **DOM vs. Framebuffer:** Web VRT relies on the Document Object Model (DOM). It knows that a specific cluster of pixels is a <button id="submit">. It can inspect the properties of that object.
- **The Embedded Void:** On the CYD, there is no DOM. There is only a framebuffer—a raw array of 16-bit color values. Traditional VRT tools cannot look at a framebuffer and say, "The button is misaligned," because they don't know what a button is without the DOM metadata.

Ouroboros solves this by using MLLMs as "Semantic Interpreters." We do not need a DOM because the MLLM uses computer vision to identify the button *conceptually*, just as a human user would. This bridges the gap between the structured web world and the unstructured framebuffer world.

---

4. The Protocol: Model Context Protocol (MCP) as a Physical Bridge

The Model Context Protocol (MCP) is the glue that binds the digital reasoning of the Agent to the physical actuation of the Enforcer and Subject. It transforms "natural language" into "voltage."

4.1 MCP Architecture for Hardware

The Ouroboros implementation utilizes a "Host-Server" topology.

- **The Host:** The AI Agent (running in a customized environment like Claude Desktop or a Python-based LangChain wrapper).
- **The Servers:** We deploy two distinct MCP servers to handle the different domains of reality.

4.1.1 The Serial-Compiler Server

This server is a wrapper around the arduino-cli and esptool binaries. It exposes the toolchain as executable functions.

Key Tools:

- list_ports(): Returns a JSON array of available serial devices.
  - *Insight:* This tool is the first check for system health. If the list is empty, the Agent knows to call the Enforcer.
- compile_sketch(source_code, fqbn): Writes the code to a temporary .ino file, invokes

arduino-cli compile, and returns the stdout (success) or stderr (compiler errors).
  - *Insight:* The Agent learns from stderr. If the compiler complains about a missing library, the Agent acts to install it or rewrite the code.
- upload_firmware(port, fqbn): Invokes arduino-cli upload.
  - *Security Note:* This is the dangerous step. A bad upload can brick the unit.
- read_serial_monitor(port, duration): Opens the serial port, captures logs for N seconds, and returns them.

4.1.2 The Enforcer Control Server

This server communicates with the Enforcer microcontroller (e.g., via a secondary USB-Serial link or direct GPIO manipulation on a Raspberry Pi host).

Key Tools:

- trigger_reset(mode):
  - mode="soft": Pulses EN Low. Standard reboot.
  - mode="hard": Pulses EN Low while holding BOOT Low. Forces Download Mode.
- power_cycle(): If the system includes a relay on the USB VBUS (5V) line, this tool cuts power entirely. This is the "Nuclear Option" for clearing latch-ups in the silicon that a simple reset cannot fix.

4.2 The JSON-RPC Bridge

The communication adheres to the JSON-RPC 2.0 specification defined by MCP. This rigorous structure prevents "hallucinated commands." The Agent cannot just make up a command; it must conform to the schema defined by the server.

Example Request (Agent to Enforcer):

JSON

```json
{
 "jsonrpc": "2.0",
 "method": "call_tool",
 "params": {
  "name": "trigger_reset",
  "arguments": {
   "mode": "hard",
   "reason": "USB_CDC_DEADLOCK_DETECTED"
  }
 },
  "id": 42
}
```

Example Response (Enforcer to Agent):

JSON

```json
{
  "jsonrpc": "2.0",
  "result": {
    "status": "success",
    "message": "Target forced into Download Mode. Waiting for enumeration."
  },
  "id": 42
}
```

This structured dialogue turns physical actions into loggable, debuggable software events. It allows the Agent to "reason" about physics using the same logic it uses for code.

---

5. The Enforcer: Engineering Physical Sovereignty

The Enforcer is not merely a component; it is a concept of "Physical Sovereignty." It ensures that the control loop always has a higher authority than the software running on the target.

5.1 The Physics of the Kill Switch

Designing the Enforcer requires careful attention to the electrical characteristics of the ESP32. We cannot simply connect a wire and hope for the best.

5.1.1 The EN Pin Circuitry

The ESP32's EN (Enable) pin is an active-low reset. On the CYD, the schematic shows an RC delay circuit:

- **Pull-Up Resistor (R_{PU}):** Typically 10kΩ, connected to 3.3V.
- **Capacitor (C_{Delay}):** Typically 1µF or 100nF, connected to GND.

This RC circuit ($T = R \times C$) ensures that when power is applied, the voltage on the EN pin rises slowly, allowing the power rails to stabilize before the chip boots.

5.1.2 The Open Drain Necessity

The Enforcer (e.g., a Raspberry Pi Pico running at 3.3V) must control this pin. However, we face a potential conflict.

- *Scenario:* The user presses the physical RESET button on the CYD (shorting EN to GND) while the Enforcer tries to drive the pin HIGH (3.3V).
- *Result:* A short circuit. Current rushes from the Enforcer's GPIO to the switch's ground connection, potentially burning out the Enforcer's output driver.

Solution: The Enforcer must configure its control pin as **Open Drain**.

- **Logic 0 (Reset):** The pin drives LOW (connects to GND), overcoming the 10kΩ pull-up on the CYD.
- **Logic 1 (Run):** The pin enters High-Impedance (Hi-Z) mode. It effectively disconnects. The CYD's internal 10kΩ resistor pulls the line back to 3.3V.

This ensures that the Enforcer can pull the trigger, but it never fights the existing circuitry.

5.2 The Bootloader Sequence Timing

Entering the ROM Serial Bootloader is a precise dance. The ESP32 samples the **BOOT** (GPIO 0) pin on the *rising edge* of the **EN** pin.

The Algorithm:

1. **T=0ms:** Enforcer drives GPIO 0 LOW. (Prepare for bootloader).
2. **T=50ms:** Enforcer drives EN LOW. (Chip is in reset).
3. **T=150ms:** Enforcer releases EN (Hi-Z). (EN voltage rises due to RC circuit).
4. **T=151ms:** The ESP32 hardware detects the rising edge on EN. It samples GPIO 0. It sees LOW. It enters Download Mode.
5. **T=250ms:** Enforcer releases GPIO 0 (Hi-Z).

If this timing is violated—for example, if GPIO 0 is released before EN rises—the chip will boot into standard execution mode (User Code), and the "unbricking" attempt will fail. The Enforcer's firmware must be written with hard real-time guarantees to ensure this sequence is chemically pure.

---

6. The Observer: Visual Truth in a Noisy World

The Observer is the final arbiter of truth. But "seeing" is difficult.

6.1 The Challenge of Screen Capture

Capturing an image of an LCD screen is fraught with technical peril.

- **Glare and Reflection:** A glossy screen acts as a mirror. If the room lights are reflected in the screen, the OCR engine might try to read the reflection of a ceiling poster instead of the UI text.
  - *Quantara Approach:* We use a **Circular Polarizer (CPL)** filter on the webcam lens, rotated to cross-polarize the light emitted by the LCD, cutting through surface glare.
- **Refresh Rate Sync:** The LCD updates at 60Hz. The webcam captures at 30Hz or 60Hz. If they are not synchronized, the camera might capture a "rolling shutter" artifact—a half-drawn frame.
  - *Quantara Approach:* The Agent is instructed to add a "Stabilization Delay" in the firmware code—a delay(500) after drawing the UI—before signalling the Observer to capture. This ensures the frame is static.

6.2 MLLM Capabilities: GPT-4o vs. Claude 3.5 Sonnet

Our research indicates a divergence in MLLM capability for this specific task.

Feature	GPT-4o	Claude 3.5 Sonnet
OCR (Text)	Excellent. High accuracy on standard fonts.	Very Good. Can struggle with extremely low-contrast pixel fonts.
Spatial Reasoning	Moderate. Can identify objects, but struggles with precise pixel alignment ("Is it 10px off?").	**Superior.** Excellent at relative positioning and detecting layout symmetry.
Artifact Detection	Good. Recognizes "glitchy" images.	**Superior.** Can describe *why* an image looks glitchy (e.g., "The RGB channels appear misaligned").
Context Window	128k tokens. Good for code history.	200k tokens. Better for holding long visual debug sessions.
Batching	**Poor.** Struggles when analyzing multiple historical screenshots at once.	Better consistency across multi-turn visual conversations.

Based on this, Ouroboros defaults to **Claude 3.5 Sonnet** for the Visual Feedback Loop, utilizing its spatial reasoning to fine-tune UI elements.

6.3 The "Hallucination of Success"

A critical function of the Observer is to verify negative space.
When an agent draws a black button on a black background, a compiler sees success. A blind agent sees success. The Observer sees nothing.
The Ouroboros prompt engineering explicitly asks the MLLM: "Describe what is NOT visible that SHOULD be visible." This forces the model to reason about the absence of features, catching contrast issues that would render a device unusable in the real world.

---

7. Quantara's Vision: The Future of Decentralized Development

Ouroboros is more than a testing rig; it is a manifesto for the future of hardware.

7.1 Local Loops and Privacy

In an era of surveillance capitalism, "The Cloud" is a liability. Sending your proprietary firmware code to a centralized server for analysis is a risk. Ouroboros is designed to be **Local-First**.

- The Agent can be swapped from a cloud MLLM (Claude) to a local open-weights model (e.g., Llama 3-V 70B) running on the engineer's GPU.
- The hardware sits on the desk, physically air-gapped from the internet if necessary (aside from the local model inference).
  This allows for "Dark Development"—the creation of sensitive, privacy-centric hardware tools in complete isolation, with the productivity benefits of AI but without the data leakage.

7.2 The Democratization of Reliability

By replacing a $20,000 dSPACE rack with a $15 CYD, a $30 webcam, and a $5 Pico Enforcer, we are lowering the barrier to entry for high-reliability engineering by three orders of magnitude.
This means a student in Lagos, a hacker in Berlin, or a boutique firm like Quantara can build devices that are verified to an industrial standard. We can run regression tests 24/7, catching edge cases that manual testing misses. We can build "Self-Healing Firmware" that iterates on itself overnight, waking up the engineer with a perfected binary in the morning.

7.3 Machines That See Their Own Code

Finally, there is the philosophical implication. Ouroboros creates a machine that possesses a rudimental form of self-awareness. It acts (writes code), it observes the consequence of that

action (sees the screen), and it modifies its behavior to align the result with its intent.
This "Closed-Loop Optical Verification" is the precursor to true machine agency. It moves us away from machines that are simply tools, and toward machines that are craftsmen. They do not just execute; they improve. And in the chaotic, noisy, broken world of embedded systems, the ability to improve is the only ability that matters.

Conclusion

The Ouroboros architecture represents a paradigm shift in embedded development. It rejects the blind trust of software simulation and the exorbitant costs of industrial emulation. Instead, it embraces the messy reality of the physical world, using vision to ground the AI agent in the truth of the hardware.

By implementing the Subject (CYD), Observer (Webcam), Agent (MCP), and Enforcer (Watchdog), we create a system that is robust, autonomous, and self-correcting. We turn the fragility of cheap hardware into a training ground for resilient code. This is the Quantara way: verified, private, and relentlessly grounded in reality.

The loop is closed. The machine can see. Now, let it build.

Appendix A: Technical Reference

A.1 Component List

- **Subject:** ESP32-2432S028R (CYD) - Resistive Touch Variant.
- **Enforcer:** Raspberry Pi Pico (RP2040) or Arduino Nano.
- **Observer:** Logitech C920 HD Pro Webcam.
- **Host:** PC running Python 3.10+ (for MCP Servers).

A.2 Wiring Diagram (Enforcer to Subject)

Signal	Enforcer Pin (Pico)	Subject Pin (CYD)	Connection Type	Note
RESET Control	GP14	EN / RST	NPN Transistor (Base)	Drive HIGH to Reset (pull EN Low).
BOOT Control	GP15	GPIO 0	NPN Transistor (Base)	Drive HIGH to enter

				Download Mode.
Serial RX	GP1 (UART0 RX)	TX0 (Header P1)	Direct	Log capture.
Serial TX	GP0 (UART0 TX)	RX0 (Header P1)	Direct	Command injection.
Ground	GND	GND	Common	**Mandatory.**

Note: If using NPN transistors (e.g., 2N2222), connect Emitter to GND, Collector to CYD Pin, and Base to Enforcer Pin via 1kΩ resistor. This creates an Open Drain equivalent circuit.

A.3 MCP Server Configuration (reference)

JSON

```json
{
 "mcpServers": {
  "serial": {
   "command": "uv",
   "args": ["run", "serial_mcp_server.py"]
  },
  "vision": {
   "command": "uv",
   "args": ["run", "vision_mcp_server.py", "--camera", "0"]
  }
 }
}
```

Sky U // Quantara
December 13, 2025