Documentation: Designing a New Instrument

1. Our Design Philosophy

This framework is built on a state-machine-based design philosophy. This approach helps manage complexity, prevent bugs, and create responsive, non-blocking instruments. The core principles are:

- **One Machine, One State:** Each instrument is represented by a single, unified `StateMachine` instance. At any given moment, the machine is in one, and only one, well-defined state (e.g., `Initializing`, `Idle`, `Moving`).
- **Separation of Data and Logic:** We strictly separate the instrument's static configuration (the *data*, like pin numbers and settings) from its dynamic behavior (the *logic*, defined in states and handlers).
- **Centralized Command Handling:** The `StateMachine` itself is responsible for managing and dispatching commands. States simply indicate *when* the machine should listen for those commands.
- **Common, Reusable Components:** We leverage a library of common states (`GenericIdle`, `GenericError`) and command handlers (`ping`, `help`) to reduce boilerplate code and ensure consistent behavior across all instruments.

Following this design process will produce a clear "spec sheet" for your new instrument. This document serves as a blueprint that a programmer—or a capable AI—can use to draft the firmware files (`__init__.py`, `states.py`, `handlers.py`) with high accuracy.

2. The Design Process: A Question-Driven Approach

The best way to design a new instrument is to answer a series of questions, moving from a high-level overview down to the specific details of each state. This Socratic, question-driven method ensures all aspects of the instrument's behavior are considered.

Step 1: High-Level Instrument Definition

Before writing any code, answer these four core questions about the instrument as a whole:

1. **What is the primary purpose of this instrument?** (e.g., "To measure temperature and humidity," "To move a robotic arm to specified coordinates.")
2. **What are the primary actions it can perform?** (e.g., "Take a reading," "Move to an X,Y coordinate," "Turn a motor on.")
3. **What data does it need to report back periodically?** (This defines your telemetry. e.g., "The current temperature," "The motor's logical position.")
4. **What are the critical failure conditions?** (e.g., "The sensor cannot be found on the I2C bus," "An endstop is hit unexpectedly.")

Step 2: Defining the Hardware Configuration

All static hardware definitions and settings belong in a single `CONFIG` dictionary in the instrument's `__init__.py` file. This creates a central "dashboard" for the instrument's physical setup.

- List every pin the microcontroller will use. Give each pin a descriptive name.
- List any key operational parameters (e.g., motor speeds, pump timings, physical dimensions).
- List any "safe limits" that the instrument must not violate.

Step 3: Defining the Command Interface

This defines the API that the host computer will use to control the instrument. For each custom command, define:

- **Function Name (`func`):** A short, verb-based name (e.g., `read_now`, `move_to`).
- **Description:** A clear, one-sentence description of what the command does.
- **Arguments (`args`):** What data, if any, the host must provide.
- **Success Condition:** What happens when the command completes successfully? (e.g., "Returns a `SUCCESS` message with the sensor data," "Transitions to the `Moving` state.")
- **Guard Conditions:** What criteria must be met for this command to be accepted? (e.g., "The machine must be homed," "The target coordinates must be within safe limits.")

Step 4: Defining the States

Now, apply the Socratic method to each individual state your instrument will have. Remember, you get `GenericIdle` and `GenericError` for free, so you only need to design the states unique to your instrument.

For each custom state, answer these five questions:

1. **Purpose:** What is this state's single, clear responsibility?
2. **Entry Actions (`enter()`):** What needs to be set up *the moment* we enter this state? (e.g., start a timer, reset a counter, turn on a pin).
3. **Internal Actions (`update()`):** What is the main work this state does on *every loop*? (e.g., check a timer, monitor a sensor, step a motor, listen for commands).
4. **Exit Events & Triggers:** How does this state know its job is finished? What causes a transition to another state? (e.g., a timer expires, a task is complete, an error is detected, an `abort` command is received).
5. **Exit Actions (`exit()`):** What cleanup is required when leaving this state to ensure the hardware is safe? (e.g., turn off a motor, reset a flag).

3. The Design Document Template

Copy the following Markdown template into a new file and fill it out for your new instrument. This completed document is the final "spec sheet."

Instrument Design: AHT20 Environmental Sensor

1. Instrument Overview

- **Primary Purpose:** To measure and report ambient temperature and relative humidity.
- **Primary Actions:** Initialize the sensor, periodically read sensor values, and perform a manual read on command.
- **Periodic Telemetry Data:** The current temperature (in Celsius) and relative humidity (in %).
- **Critical Failure Conditions:** The AHT20 sensor is not detected on the I2C bus during initialization.

2. Hardware Configuration (`CONFIG` dictionary)

(Note: The AHT20 uses the board's default I2C pins, so we don't need to define them explicitly in this simple case.)

```
AHT20_CONFIG = {
    # No custom pin definitions needed for this simple sensor.
    # A more complex device would list all pins here.
}
```

3. Command Interface

`func` Name	Description	Arguments	Success Condition	Guard Conditions
`read_now`	Immediately reads the sensor and returns the values.	None	Returns a `SUCCESS` message with a payload containing the temperature and humidity.	None.

4. State Definitions

State: `Initialize`

1. **Purpose:** To connect to the AHT20 sensor via the I2C bus.
2. **Entry Actions (`enter()`):**
    - Attempt to create an `I2C` object for the board.
    - Attempt to instantiate the `adafruit_ahtx0.AHTx0` sensor object using the I2C bus.
    - Attach the sensor object to the `machine` instance (e.g., `machine.sensor`).
3. **Internal Actions (`update()`):** None. This state should transition immediately.
4. **Exit Events & Triggers:**
    - **On Success:** The sensor is instantiated without error. Trigger: Transition to `Idle`.
    - **On Failure:** An exception is raised (e.g., `ValueError` if the sensor is not found). Trigger: Set an `error_message` flag and transition to `Error`.
5. **Exit Actions (`exit()`):** None.