# Architectural Analysis & Integration Strategy for PyRPL in PyMoDAQ

A Technical Presentation for PyMoDAQ Developers

## Agenda

1. **The Challenge**: Integrating PyRPL with PyMoDAQ.
2. **Core Architectures**: A tale of two frameworks.
3. **The Fundamental Incompatibility**: Why direct integration fails.
4. **Solution 1**: The IPC Wrapper (Pragmatic & Immediate).
5. **Solution 2**: The Native Plugin (Ideal & Long-Term).
6. **Recommendations & Path Forward**.

## The Challenge: A Tale of Two Frameworks

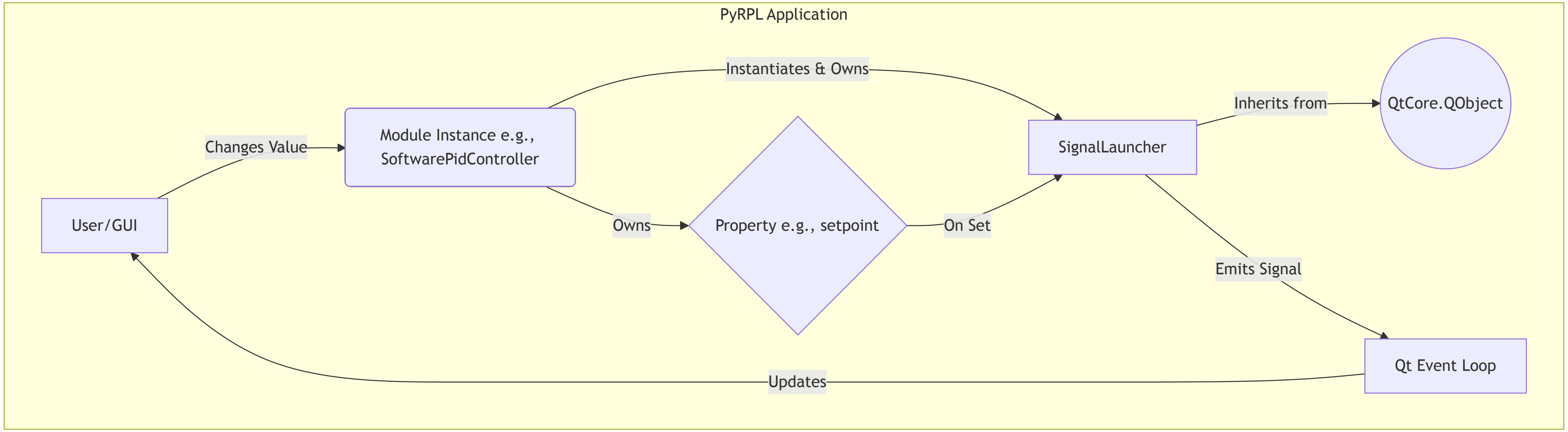
**Goal**: Integrate the **PyRPL** instrument framework as a standard **PyMoDAQ** plugin.

- **PyRPL**: A powerful, monolithic Qt application that turns a Red Pitaya into a multi-function lab instrument (Scope, PID, Lock-in).
- **PyMoDAQ**: A modular, multi-threaded data acquisition platform.

**Problem**: Attempting to instantiate PyRPL objects within a PyMoDAQ plugin causes an immediate, fatal crash related to Qt thread affinity.

# PyRPL Architecture: A Qt Application

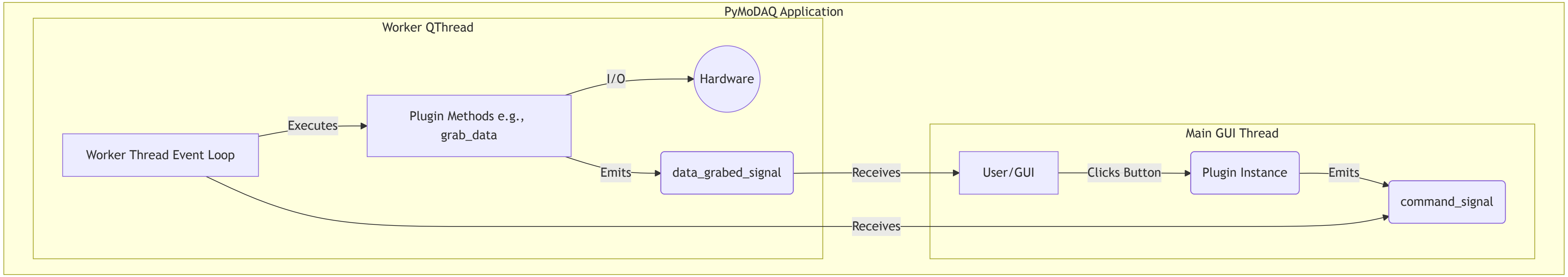
PyRPL is not a library; it's an application. Its core components are deeply integrated with `QtCore.QObject` to drive its GUI.




**Key takeaway**: Every hardware/software `Module` in PyRPL creates and owns a `QObject` ( `SignalLauncher` ) to handle signals and events.

# PyMoDAQ Architecture: Multi-Threaded by Design

PyMoDAQ ensures a responsive GUI by offloading all hardware communication to a dedicated worker `QThread`.




**Key takeaway**: Plugin methods like `ini_detector()` and `grab_data()` execute in a **worker thread**, not the main GUI thread.

## The Incompatibility: A QObject in the Wrong Thread

The conflict is unavoidable. PyMoDAQ tries to instantiate a PyRPL `Module` inside the worker thread. This violates Qt's fundamental threading rules.

Instantiating a `QObject` in a worker thread that belongs to the main thread's ecosystem is forbidden. **This is not a bug; it is a fundamental design conflict.**

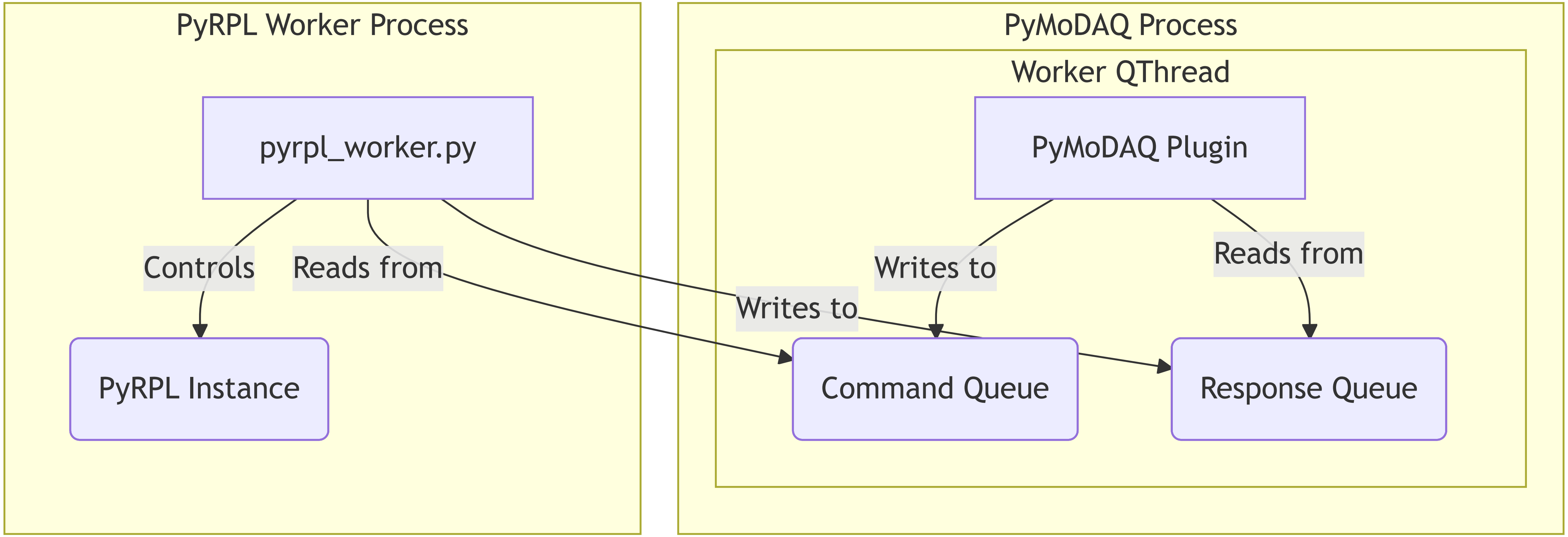
## The Incompatibility: The Smoking Gun

The evidence is in `pyrpl/pyrpl/modules.py`, where every `Module` instantiates a `QObject`.

```
from qtpy import QtCore

class SignalLauncher(QtCore.QObject): // It's a QObject!
    ...

class Module(...):
    _signal_launcher = SignalLauncher

    def __init__(self, parent, name=None):
        // This line runs in the PyMoDAQ worker thread!
        self._signal_launcher = self._signal_launcher(self)
```

## Solution 1: The IPC Wrapper (Pragmatic Fix)

Run PyRPL in a separate, isolated process and communicate with it via Inter-Process Communication (IPC). This respects the boundaries of both frameworks.




- **Pros**: 100% stable, full feature access, relatively quick to implement.

## Solution 2: Native Integration (Ideal Fix)

A long-term strategy to create a "pure" PyMoDAQ plugin by porting PyRPL's core logic.

### Three-Phase Strategy:

1. **Build a Thread-Safe API**: Create a new `RedPitayaAPI` class that encapsulates the raw TCP socket communication and uses `threading.Lock` to make memory access thread-safe.
2. **Port DSP Algorithms**: Extract the mathematical logic from PyRPL's modules (e.g., the PID algorithm) and place it in new, plain Python classes that use the thread-safe API.
3. **Create the Native Plugin**: Build a standard PyMoDAQ plugin that uses these new, thread-safe components. They are simple Python objects and will work correctly in the worker thread.

# Native Integration: Phase 1 Example

Create a new, framework-agnostic, thread-safe API.

```
# In a new file: RedPitayaAPI.py
import socket
import threading

class RedPitayaAPI:
    def __init__(self, hostname):
        self._socket = socket.socket(...)
        self._socket.connect((hostname, 2222))
        self._lock = threading.Lock() # The key to thread safety

    def read_memory(self, address, length):
        with self._lock:
            # ... sends 'r' command ...
            pass

    def write_memory(self, address, data):
        with self._lock:
            # ... sends 'w' command ...
```

# Native Integration: Phase 2 Example

Extract the pure algorithm, separating it from PyRPL's Qt-dependent structure.

### Before: PyRPL's `software_pid.py`

```
class SoftwarePidLoop(PlotLoop): # Inherits from Qt-related classes
    def loop(self):
        # ... complex logic mixed with GUI plotting and property access ...
        error = self.input - self.parent.setpoint
        self.parent._ival += self.parent.i * dt * 2.0 * np.pi * error
        # ...
```

### After: A new, clean `dsp_pid.py`

```
class NativePID:
    def __init__(self, api: RedPitayaAPI, config: dict):
        self.api = api
        self.config = config
        self._ival = 0
```

## Recommendations & Path Forward

We have two clear, viable paths. They are not mutually exclusive.

1. **Short-Term (Now)**: Implement the **IPC Wrapper**. This delivers a fully-featured, stable plugin quickly. It solves the immediate problem correctly.
2. **Long-Term (Future)**: Begin the **Native Integration** project. This is a larger effort but results in a cleaner, more performant, and more maintainable "pure" PyMoDAQ plugin. It represents the ideal architectural solution.

**Proposal**: Develop the IPC plugin now. Use the native integration strategy as the roadmap for a future, second-generation plugin.

# Q&A

**Discussion & Questions

<script type="module"> import mermaid from 'https://cdn.jsdelivr.net/npm/mermaid@10/dist/mermaid.esm.min.mjs'; mermaid.initialize({ startOnLoad: true }); </script>