PyRPL Integration: A Real-Time FPGA Control Module for PyMoDAQ

Brian Squires

University of North Texas

2025-09-30

Agenda

[.incremental]

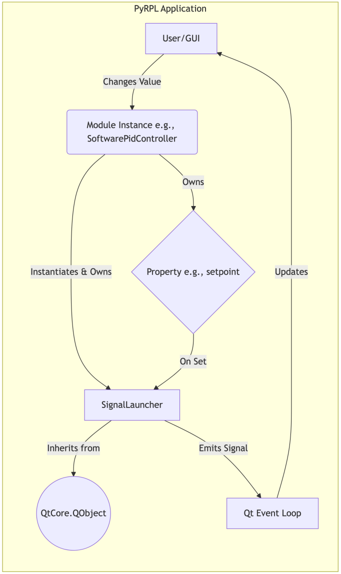
1. **The Challenge**: Integrating PyRPL with PyMoDAQ.
2. **Core Architectures**: A tale of two frameworks.
3. **The Fundamental Incompatibility**: Why direct integration fails.
4. **Solution 1**: The IPC Wrapper (Pragmatic & Immediate).
5. **Solution 2**: The Native Plugin (Ideal & Long-Term).
6. **Recommendations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

The Challenge: A Tale of Two Frameworks

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

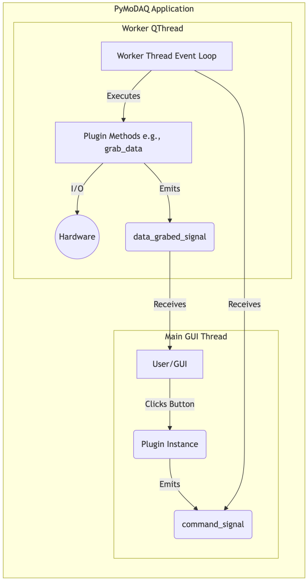
PyRPL is not a library; it's an application. Its core components are deeply integrated with `QtCore.QObject` to drive its GUI.



Key takeaway: Every `Module` in PyRPL creates and owns a `QObject` 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 execute in a **worker thread**, not the main GUI thread.

The Incompatibility

A QObject in the Wrong Thread

The conflict is unavoidable: PyMoDAQ executes hardware logic in a worker thread, but PyRPL's hardware objects are `QObjects` that expect to be in the main GUI thread.
This is not a bug; it is a fundamental design conflict.

The Incompatibility

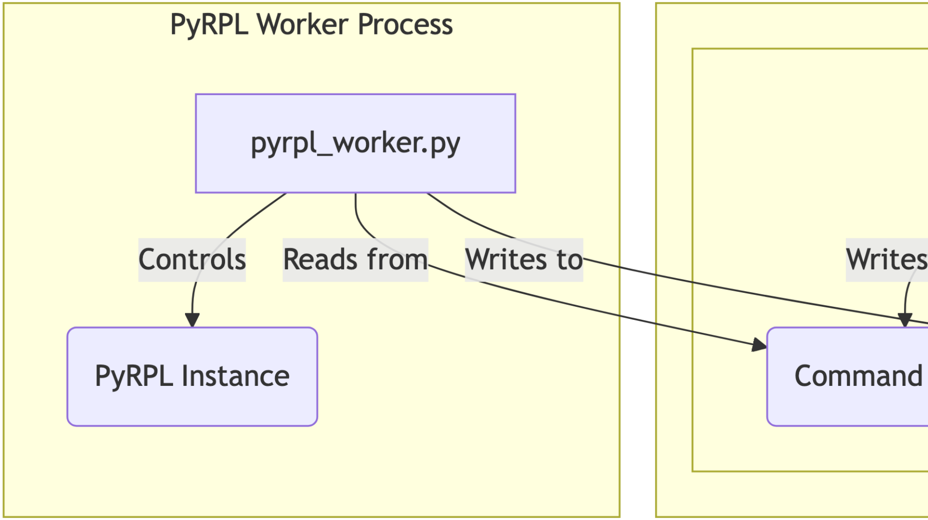
The Smoking Gun

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

Run PyRPL in a separate, isolated process and communicate with it via Inter-Process Communication (IPC).



Solution 2: Native Integration

A long-term strategy to create a “pure” PyMoDAQ plugin by porting PyRPL’s core logic.

Three-Phase Strategy:

[.incremental]

1. **Build a Thread-Safe API**: Create a new `RedPitayaAPI` class.
2. **Port DSP Algorithms**: Extract mathematical logic into plain Python objects.
3. **Create the Native Plugin**: Build a standard PyMoDAQ plugin using the new, thread-safe components.

Native Integration: Phase 2 Example

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

Before: PyRPL

```
class SoftwarePidLoop(PlotLoop)
    def loop(self):
        # ...
        error = self.input - \
                self.parent.set
        self.parent._ival +=\
                self.parent.i *
                2.0 * np.pi * e
        # ...
```

After: Native PyMoDAQ

```
class NativePID:
    def __init__(self, api, cfg
        self.api = api
        self.cfg = cfg
        self._ival = 0

    def execute(self):
        inp = self.api.read(...
        err = inp - self.cfg['s
        # ... pure DSP logic ..
        self.api.write(..., out
```

Recommendations & Path Forward

We have two clear, viable paths that are not mutually exclusive:

[.incremental]

1. **Short-Term (Now)**: Implement the **IPC Wrapper**.
   - Delivers a fully-featured, stable plugin quickly.
2. **Long-Term (Future)**: Begin the **Native Integration** project.
   - Results in a cleaner, more performant, and maintainable "pure" PyMoDAQ plugin.

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

Q&A

Discussion & Questions “‘