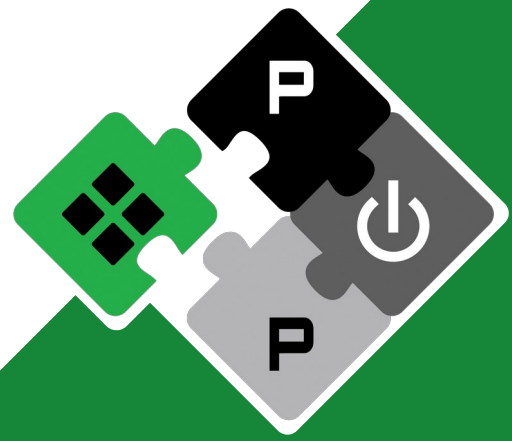
ETH zürich

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

BioGUI: An Open-Source, Device-Agnostic GUI for Biosignal Acquisition and Processing

Mattia Orlandi mattia.orlandi@unibo.it
mattia.orlandi.1@ulaval.ca



@pulp_platform
pulp-platform.org
youtube.com/pulp_platform

PULP Platform
Open Source Hardware, the way it should be!

What is BioGUI?

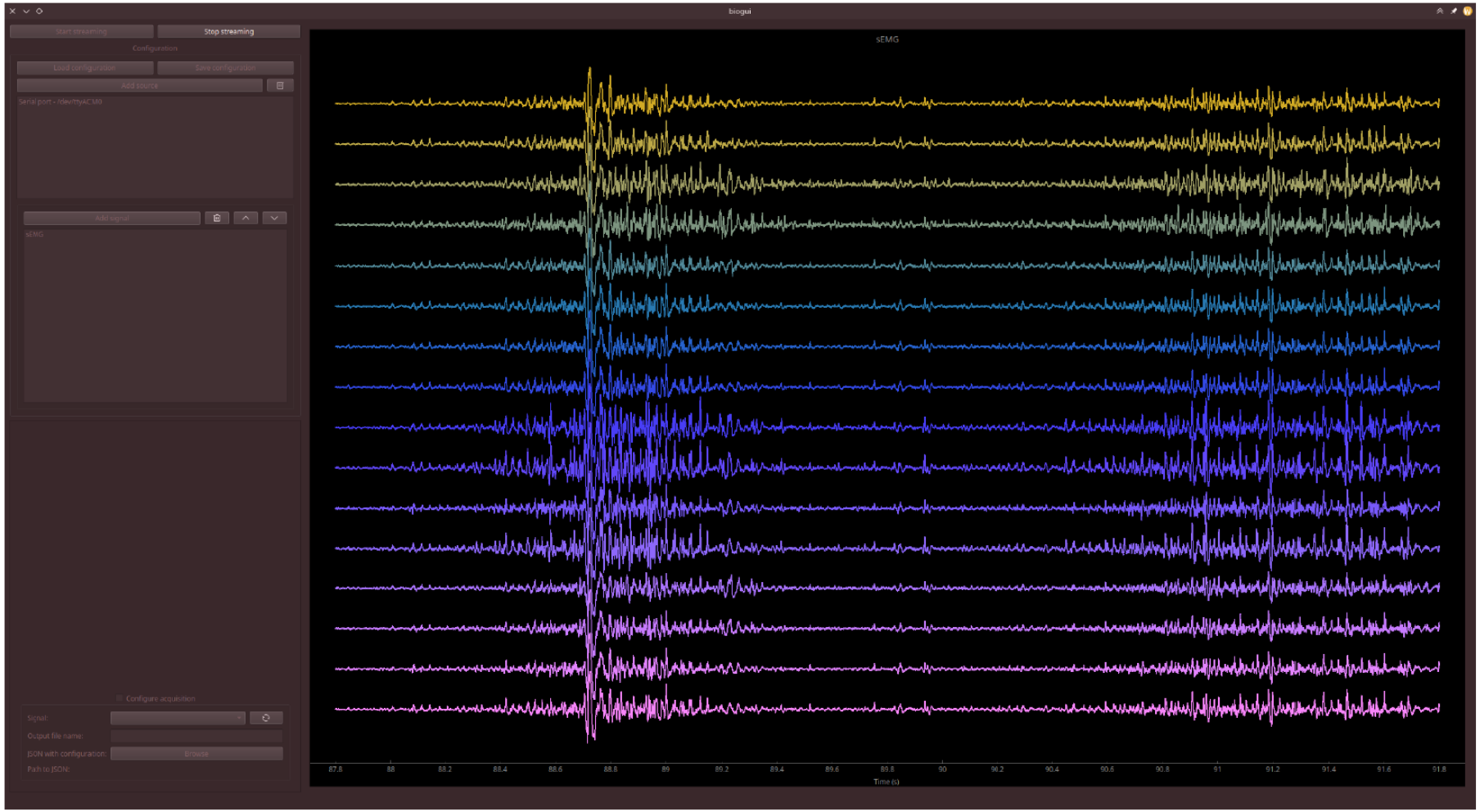
- **BioGUI is an open source framework for biosignal acquisition**
  - Open source → anyone can contribute
  - Based on Python an Qt framework → accessible to researchers without a software engineering background
  - Device-agnostic: all the device-specific details are contained in a simple Python interface → fast prototyping
- **Centralized program for all the devices you're working with**
  - both research and commercial devices
  - any time-series-like signal (biopotentials, PPG, etc.), as well as ultrasound
- **It can be easily integrated in custom acquisition and processing pipelines**
  - Write the Python interface once
  - The BioGUI displays the data, stores it, and forwards it to other processes (e.g., ML-based pipelines or control tasks)

Timeline

- **[End of 2023] The BioGUI is born**
  - At the time, we were using the BioWolf16, a custom PULP board, for acquiring sEMG signals
  - Need of a tool to visualize the signals in real time, store them, and possibly process them using ML
    - Python for overall logic → code easily interpretable even with no software engineering background
    - Qt for UI → it also provides async APIs for serial ports, sockets, bluetooth, etc.
  - → Basic GUI in PySide6 (official Python bindings for Qt)

Timeline

- **[End of**
  - At th
    signa
  - Neec
    using
    -
    -
  - → Ba



M. Orlandi et al., "Real-Time Motor Unit Tracking From sEMG Signals With Adaptive ICA on a Parallel Ultra-Low Power Processor," in IEEE Transactions on Biomedical Circuits and Systems, vol. 18, no. 4, pp. 771-782, Aug. 2024

Timeline

- **[End of 2023] The BioGUI is born**
  - At the time, we were using the BioWolf16, a custom PULP board, for acquiring sEMG signals
  - Need of a tool to visualize the signals in real time, store them, and possibly process them using ML
    - Python for overall logic → code easily interpretable even with no software engineering background
    - Qt for UI → it also provides async APIs for serial ports, sockets, bluetooth, etc.
  - → Basic GUI in PySide6 (official Python bindings for Qt)
- **[2024] Extension to other devices**
  - At some point, we were working with other custom PULP boards (GAPWatch, BioGAP) and with other signals (ECG, PPG, etc.)
  - Could we reuse the GUI?

Timeline

- **Most of the logic for data reading is device-agnostic, and can be abstracted away, as long as you know**
  - Which bytes to send to the device to start and stop the communication
  - How many bytes to read
  - How to decode those bytes into actual signals
- **We moved all the device-specific details into a so-called *interface file***
  - The interface file is a regular Python file with some predefined fields (the start and stop commands, the decode function, etc.)
  - The file is parsed by the GUI and used to communicate with the device, whereas all the downstream logic is device-agnostic

```
packetSize: int | list[tuple[int, int]]

startSeq: list[bytes | float]

stopSeq: list[bytes | float]

sigInfo: dict

def decodeFn(data: bytes)
  -> dict[str, np.ndarray]
```

Timeline

- **[2024/2025] Simultaneous streaming from multiple devices, and support of commercial devices**
  - Each device is handled by a dedicated thread → BioGUI can now stream from multiple devices at once, whose synchronization is handled via timestamps
  - Interface files for commercial products:
    - OTBioelettronica Sessantaquattro+ for HD-sEMG[1];
    - MANUS Quantum glove for joint angles[2]
- **[2025/2026] Support of ultrasound data, and header-based decoding**
  - Together with PULP team in ETH Zurich, we added support for ultrasound data and for WULPUS, a wearable device for ultrasound[3]
  - Instead of being constrained by a single packet size for every signal, the BioGUI now supports dynamic packet sizes depending on the header it receives (e.g., a packet with N bytes for EMG, and a separate packet with M bytes for IMU data)

1. M. Orlandi, et al., "Neural Strategies for Upper Limb Movements: Motor Unit Control during Dynamic Contractions at Increasing Speeds," EMBC 2025
2. G. Spacone et al., "Wearable and Ultra-Low-Power Fusion of EMG and A-Mode US for Hand-Wrist Kinematic Tracking," BioCAS 2025
3. G. Spacone et al., "A Wearable Multimodal Ultrasound+Inertial System for Real-Time Virtual Reality Interaction," arxiv.org/abs/2606.17741"

Outline

1. Qt’s key principles

- The event loop
- Notification and communication mechanisms
- Best practices

2. BioGUI description

- Architecture
- Execution flow
- Demo and hands-on

The Event Loop

- **Naive implementation:** one could implement a basic “BioGUI workflow” as in this pseudo-code
  - Manual loop in main
  - Read bytes from source (e.g., socket or serial port) and accumulate them in a buffer
  - Once buffer is full, extract the data and use it for plotting, data saving, etc.
- **Issues:**
  - Rigid execution flow: one must procedurally wait for data, then draw the plot, then save, etc.
  - Blocking operations: if the socket/serial port hangs for some reason, the whole application freezes

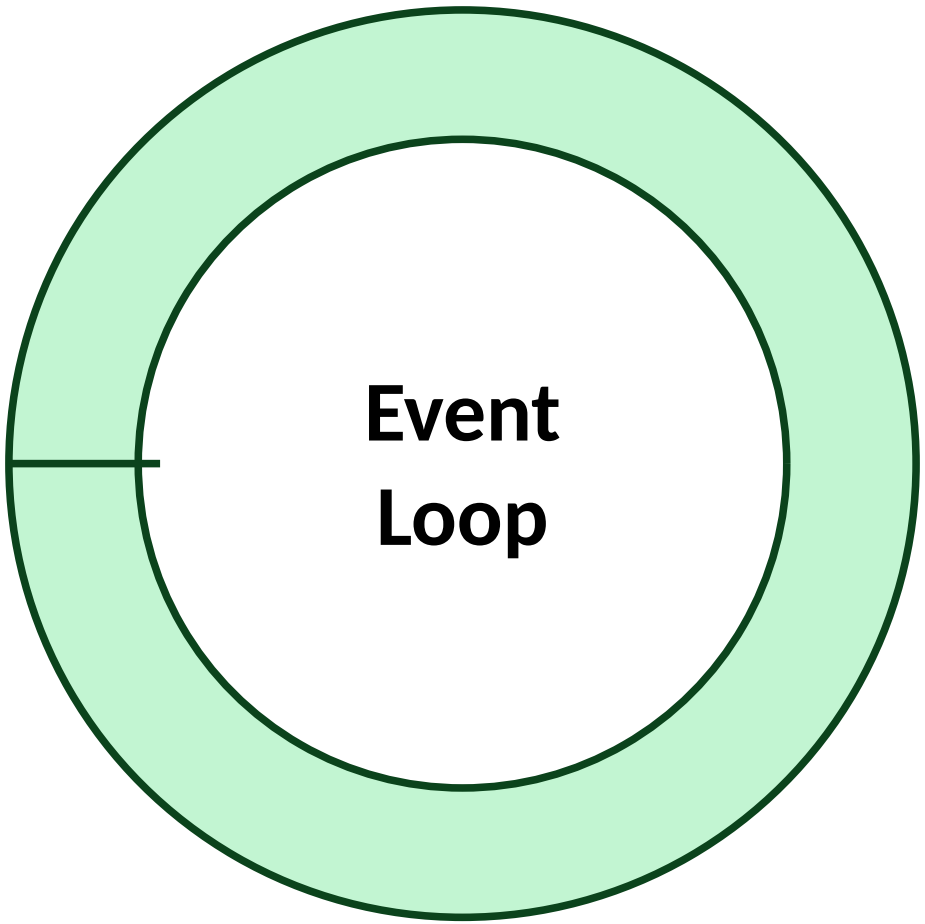
```
if __name__ == “__main__”:
  buffer = []
  # Loop
  while True:
    # Read N bytes from socket
    while True:  # blocking
      cur_data = sock.recv(N)
      buffer.extend(cur_data)
      if len(buffer) >= N:
        break
    # Get data
    data = buffer.pop(0, N-1)

    # Update plots
    # ...

    # Store data to file
    # ...
```

The Event Loop

- **Qt approach:** one needs to instantiate a “QApplication” and call the “exec()” method
  - Under the hood, the Qt engine starts an event loop
  - Once an “event” happens, it is added to a queue
  - Every time the loop repeats, the Qt engine consumes the events in the queue
  - The event loop is implemented in C++ → much more efficient than anything you could implement in bare Python

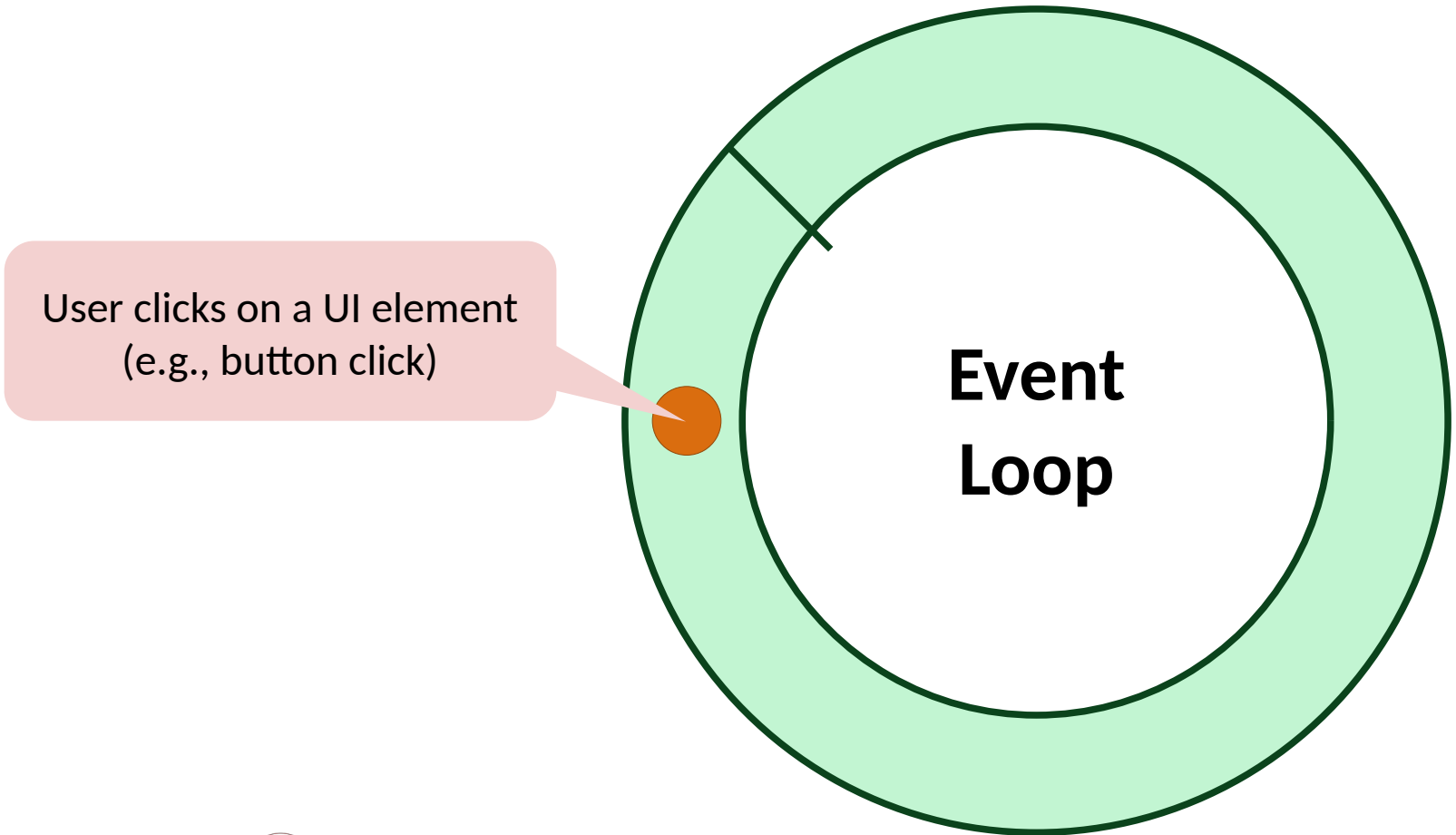
```
if __name__ == “__main__”:
    app = QApplication()
    sys.exit(
        app.exec()  # start event loop
    )
```

The Event Loop



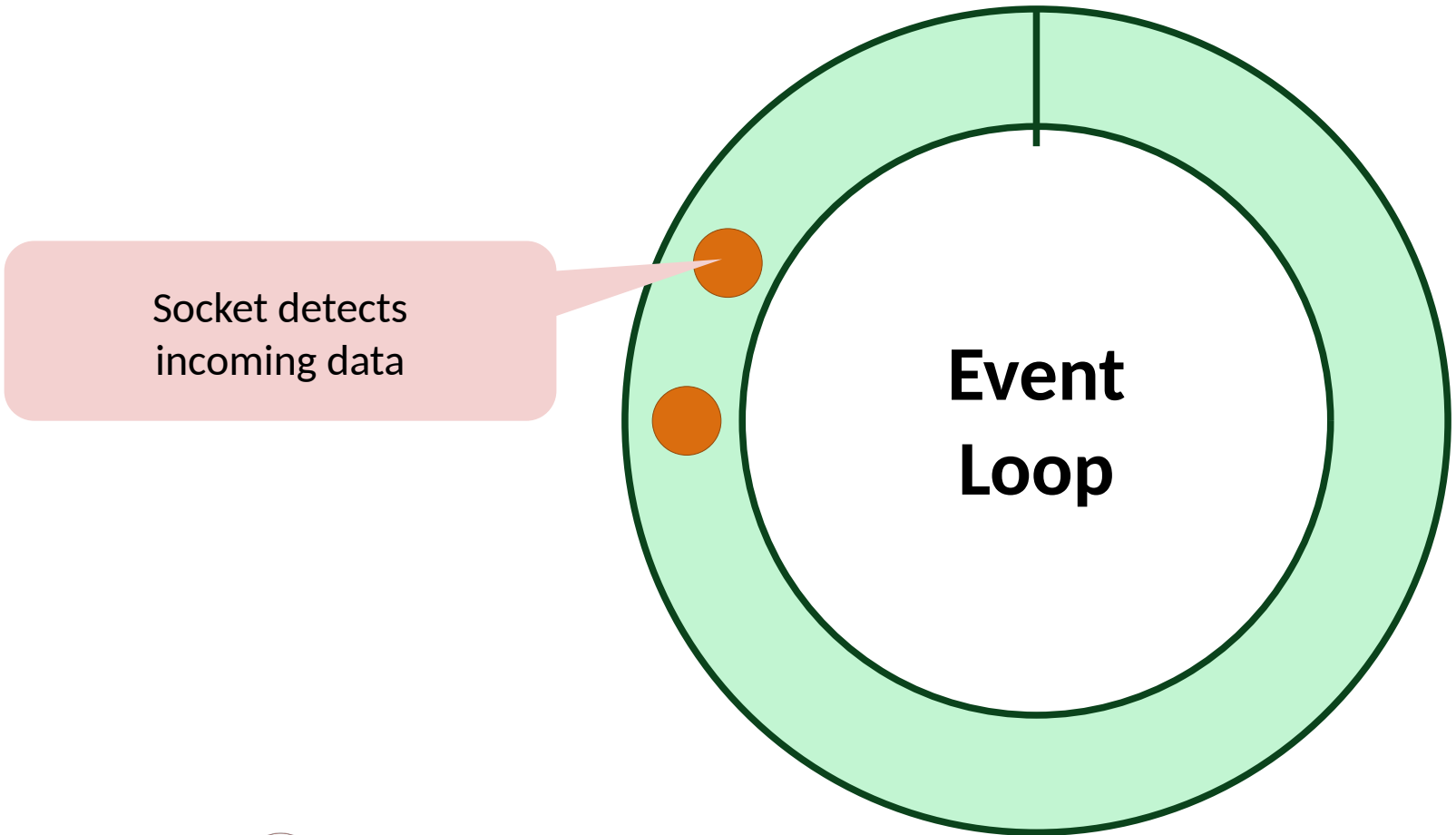
The Event Loop

- Events can be generated by user interaction with the UI…



The Event Loop

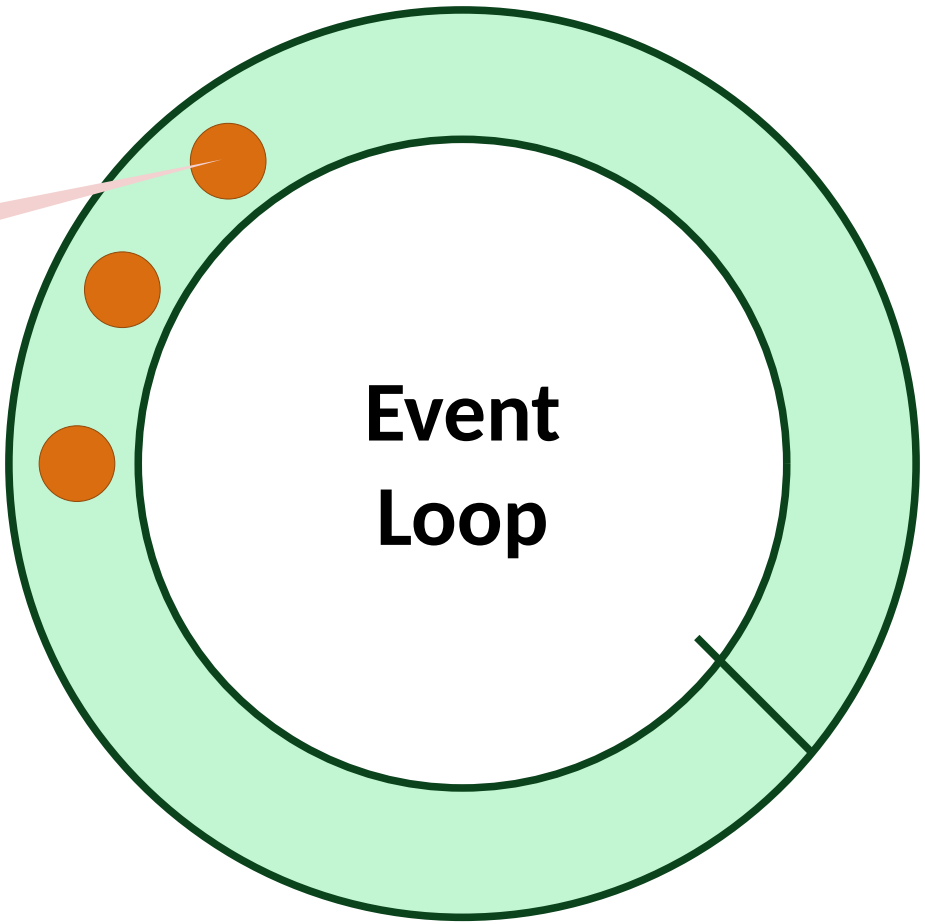
- … they can be “external” events…



The Event Loop

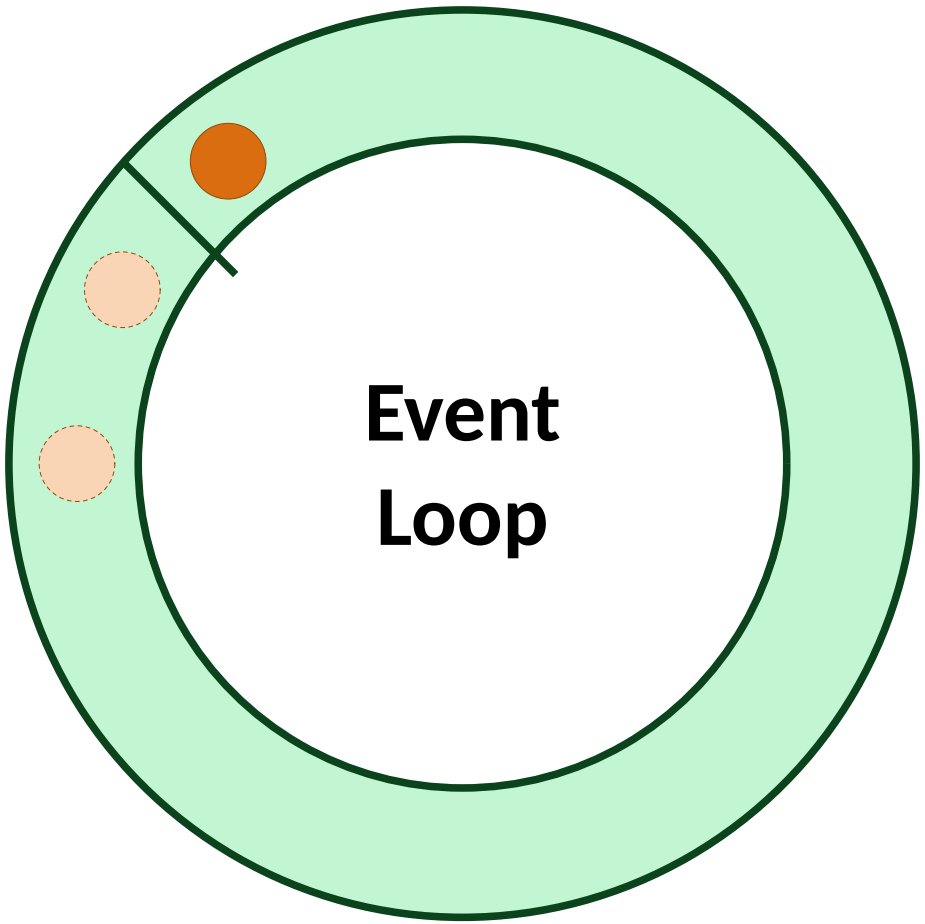
- … or even “internal”

My function finishes computing something and notifies the result



The Event Loop

- Cyclically, the Qt engine consumes all the events in the queue



Signals and Slots

- **How can we define what happens in response to an event?**
  - Signals and slots mechanism
- **Signals**:
  - Signals generate events
  - In Python, they are declared as class attributes (the class must inherit from “QObject”)
  - To generate the event, one calls the “emit()” method
  - The signal can carry data (both basic and complex types)

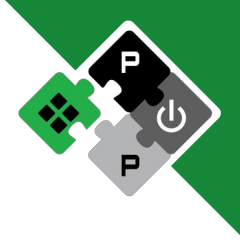
```python
class MyClassA(QObject):
  my_signal = Signal(int)

  def __init__(self, parent=None):
    # Keep a parent/children hierarchy
    super().__init__(parent)

    self._counter = 0
    self.button.clicked.connect(
      self.update_counter
    )  # wire click to update_counter

  @Slot()
  def update_counter(self):
    self._counter += 1

    # Emit event with click count
    self.my_signal.emit(self._counter)
```

Signals and Slots

- **Slots:**
  - Slots indicate the callback functions invoked when an event happens
  - In Python, one uses the “@Slot” decorator, together with the type

```
class MyClassB(QObject):

  def __init__(self, parent=None):
    super().__init__(parent)

    # obj_b will be obj_a's parent
    self._obj_a = MyClassA(self)

    self._obj_a.my_signal.connect(
      self.print_counter
    )  # wire my_signal to print_counter

  @Slot(int)
  def print_counter(self, count: int):
    print(count)
```

Side Note: QObjects and Hierarchies



- **QObjects are actually two things at once**
  - The Python object
  - The underlying Qt object in C++
- **Deallocation is tricky**
  - Before deleting a QObject in Python with “del”, one must call “deleteLater()”
  - This way, the Qt engine “knows” it has to free the memory of the C++ object
- **The importance of hierarchies**
  - When the parent is correctly deleted, all of its children are deleted too → fewer risks of memory leaks

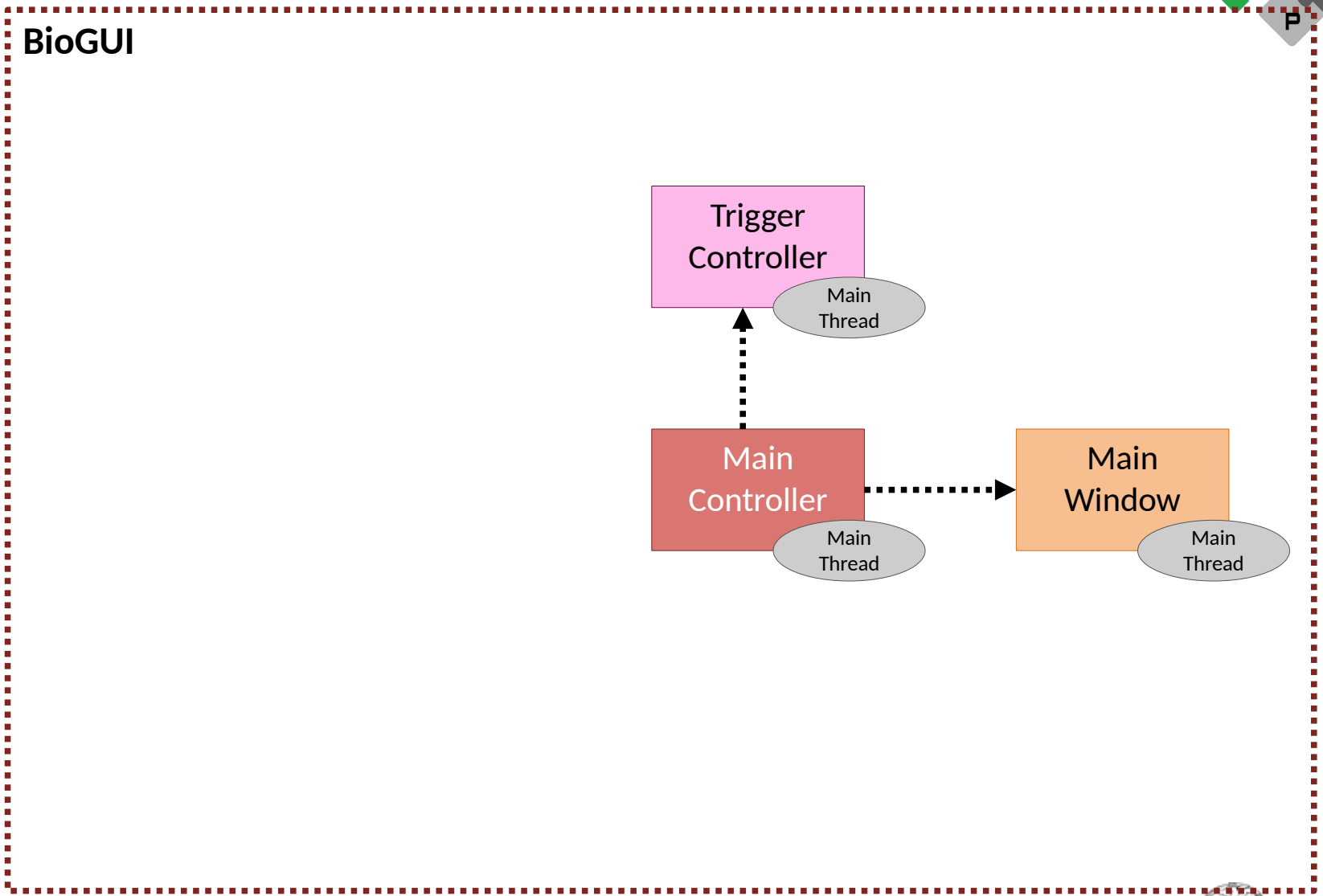
Wrap Up

- **Pros**
  - Non-blocking logic: UI stays responsive even when waiting for events to happen
  - De-coupled execution flow: instead of coding in sequence data reception, plotting, etc., one just wires up signals and slots, and lets the event loop go
- **Cons**
  - The event loop is still a loop: if it's overwhelmed by too many events, or if a slot includes blocking code, it will still slow down or freeze
- **Best practices**
  - In general, fewer events with bigger payload are better than more events with smaller payload
  - Use QThreads where appropriate:
    - each QThread spawns an independent event loop
    - different QThreads can still communicate via signals and slots
    - in general, use the main thread for UI and basic operations, and QThreads for IO operations (e.g., QThread for receiving data from socket or serial port)

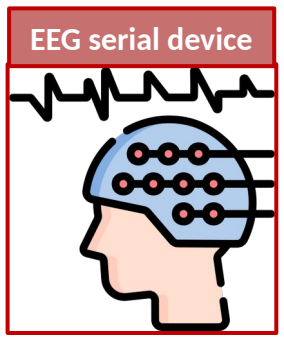
BioGUI Architecture

- **Code organization**
  - `biogui/controllers/`
    - `main_controller.py`: orchestrate everything (UI update, spawning of other components)
    - `streaming_controller.py`: controls streaming from one device by spawning a `DataSourceWorker`, and file writing via a `FileWriterWorker`; if multiple devices are added, the `MainController` spawns one `StreamingController` for each
    - `module_controller.py`: for plug-and-play modules
  - `biogui/data_sources/`
    - `base.py`: abstract base class for `DataSourceWorker` and `DataSourceConfigWidget`
    - `serial.py`: concrete `DataSourceWorker` for serial ports
    - `tcp.py`: concrete `DataSourceWorker` for TCP sockets
    - `...`
  - `biogui/modules/`
    - `trigger.py`: plug-and-play module for trigger generation: during acquisition, customizable visual cues are shown to the user (e.g., “hand close”, “hand open”, “look left”, etc.)
    - `forward.py`: plug-and-play module for forwarding data to other processes (e.g., send window to another script running ML inference, perform control tasks)
  - `biogui/platforms/`: list of curated hardware platforms with interface files

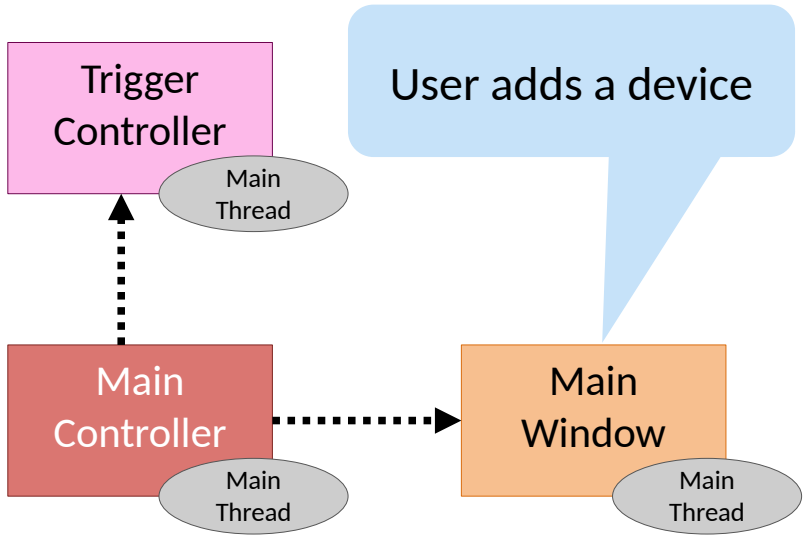
BioGUI Flow


BioGUI
Trigger Controller
Main Thread
Main Controller
Main Thread
Main Window
Main Thread

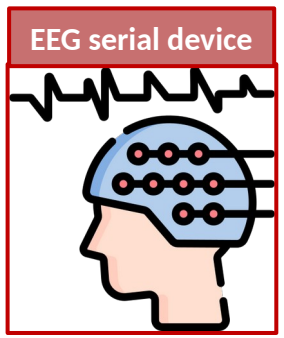
BioGUI Flow

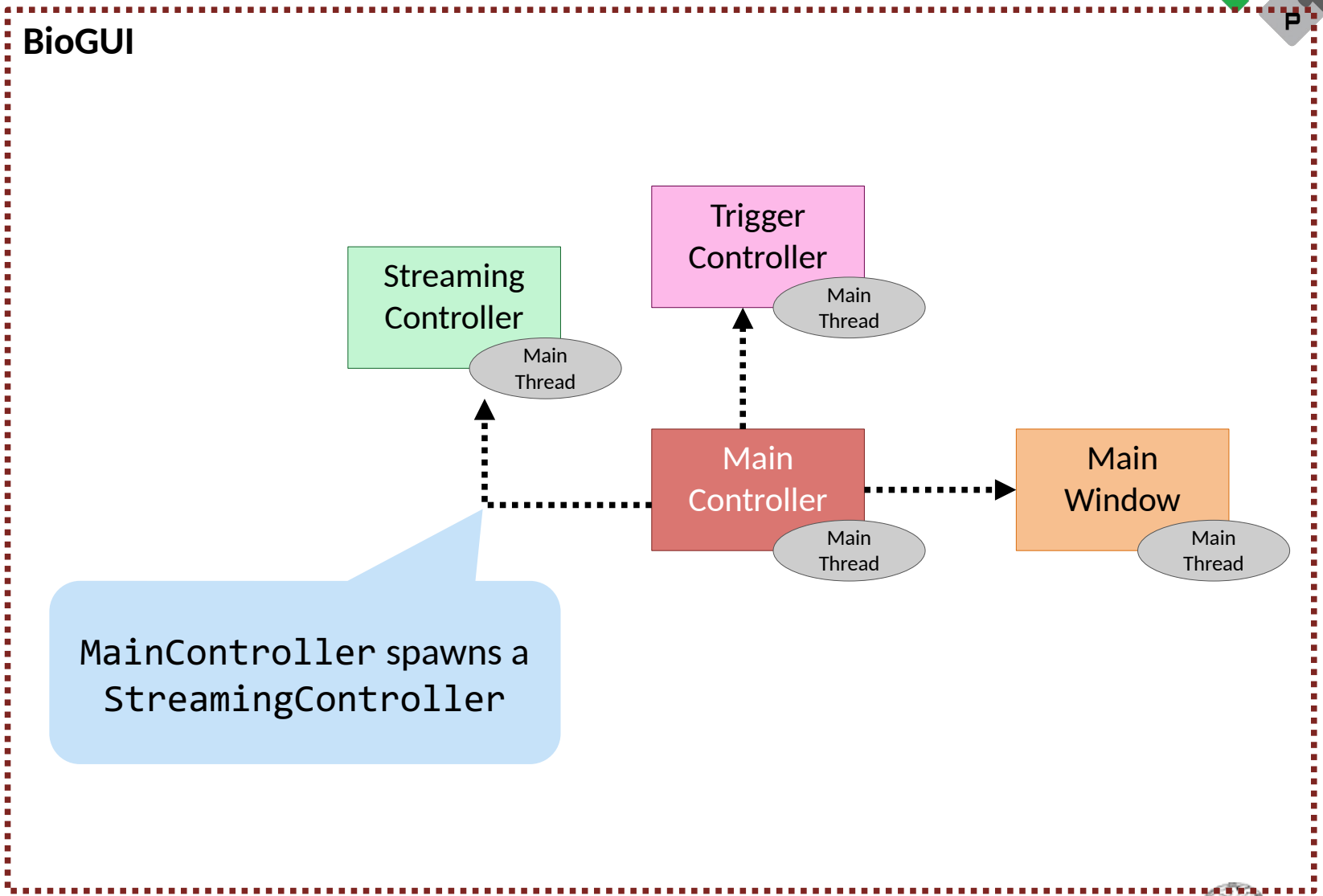


BioGUI

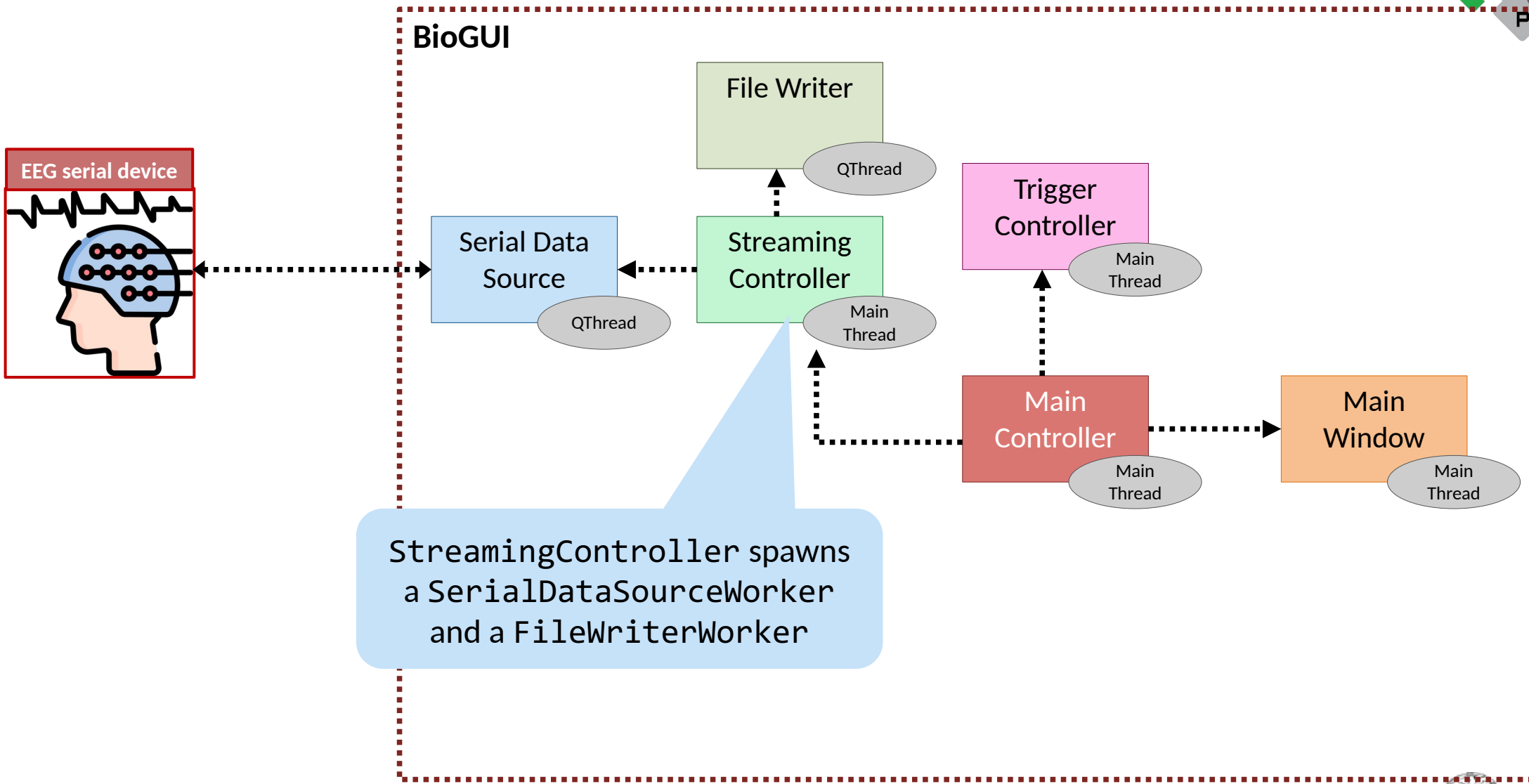


BioGUI Flow

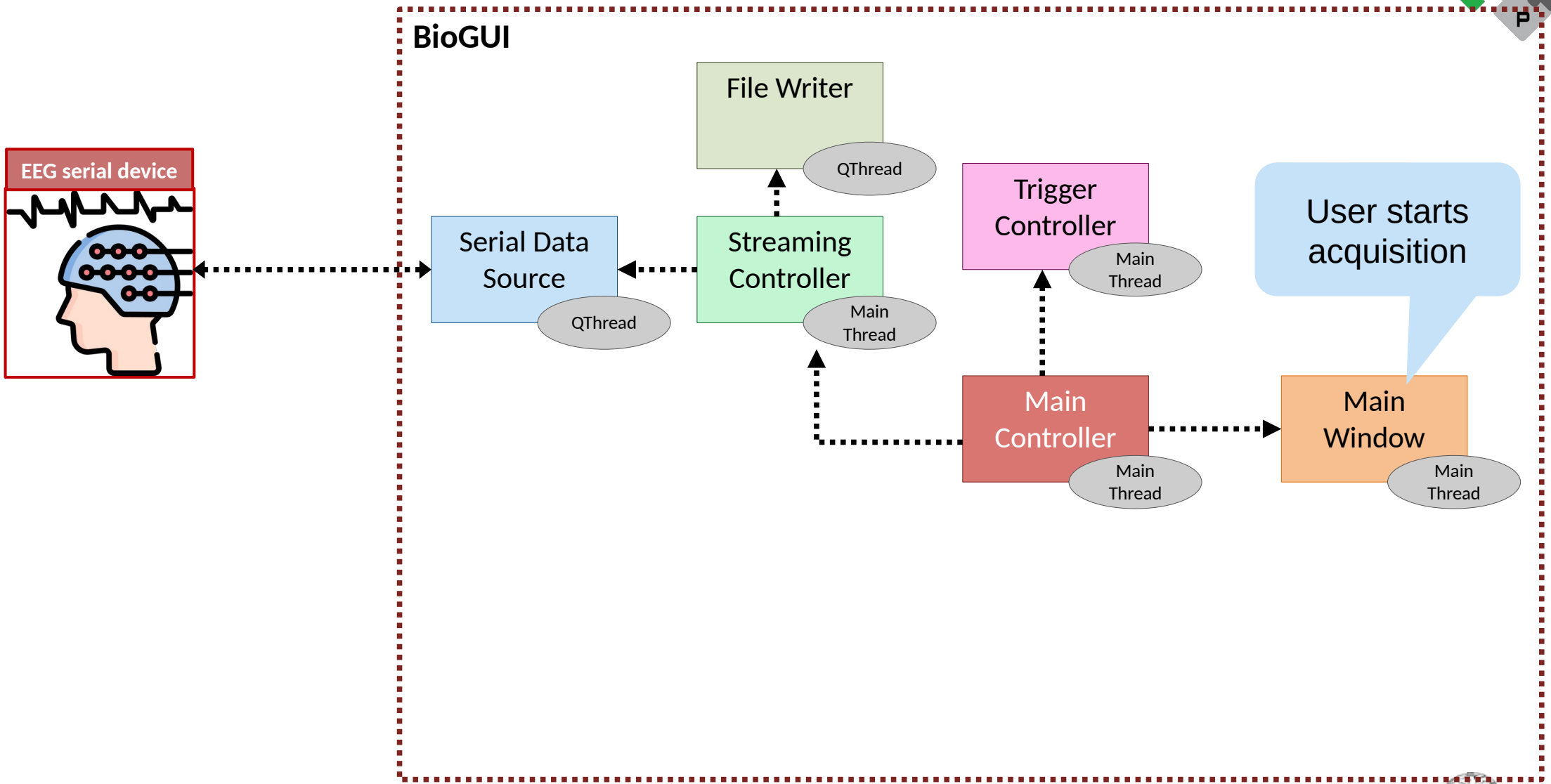

EEG serial device


BioGUI
Streaming Controller
Main Thread
Trigger Controller
Main Thread
Main Controller
Main Thread
Main Window
Main Thread
MainController spawns a StreamingController

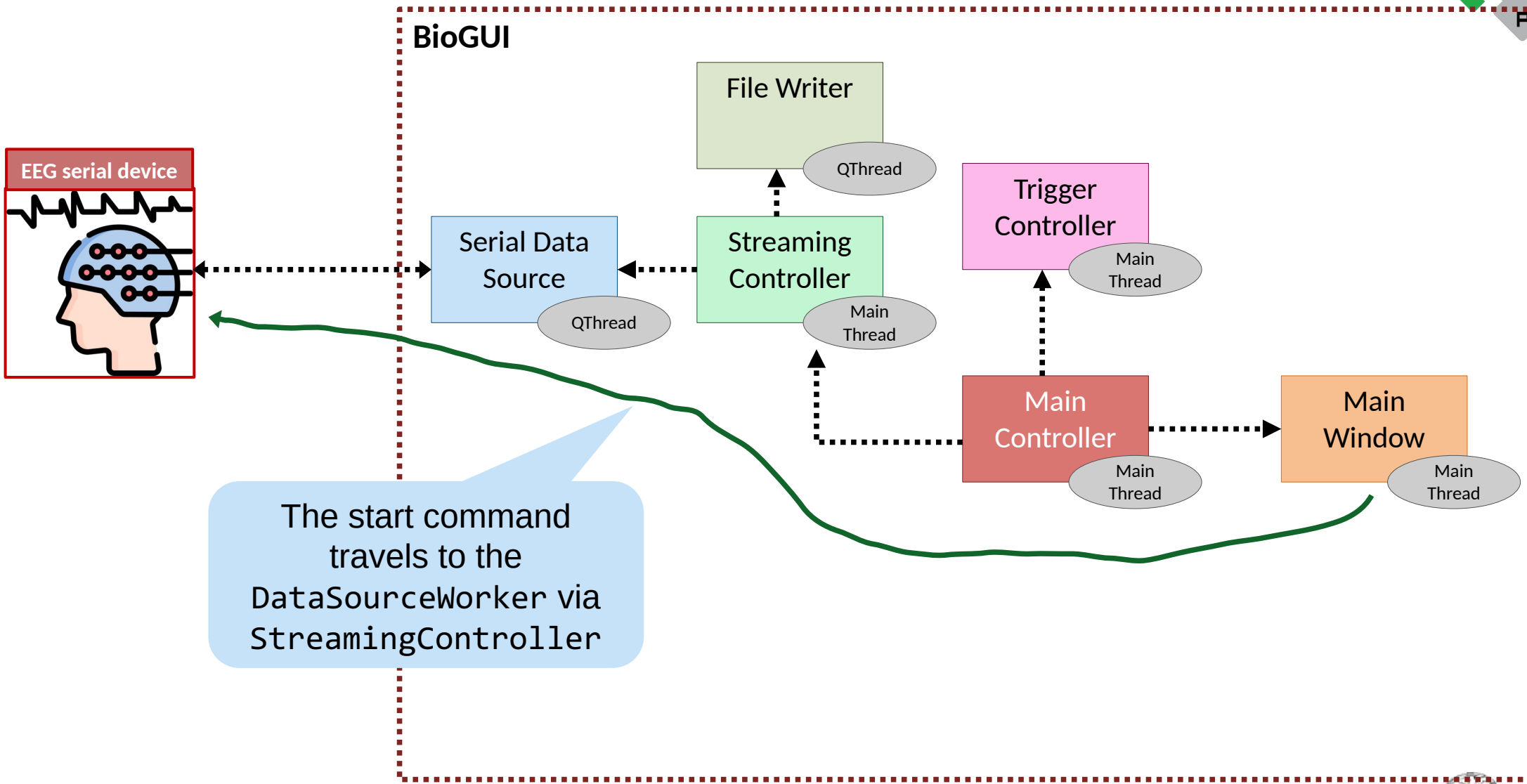
BioGUI Flow


BioGUI
File Writer
QThread
EEG serial device
Serial Data Source
QThread
Streaming Controller
Main Thread
Trigger Controller
Main Thread
Main Controller
Main Thread
Main Window
Main Thread
StreamingController spawns a SerialDataSourceWorker and a FileWriterWorker

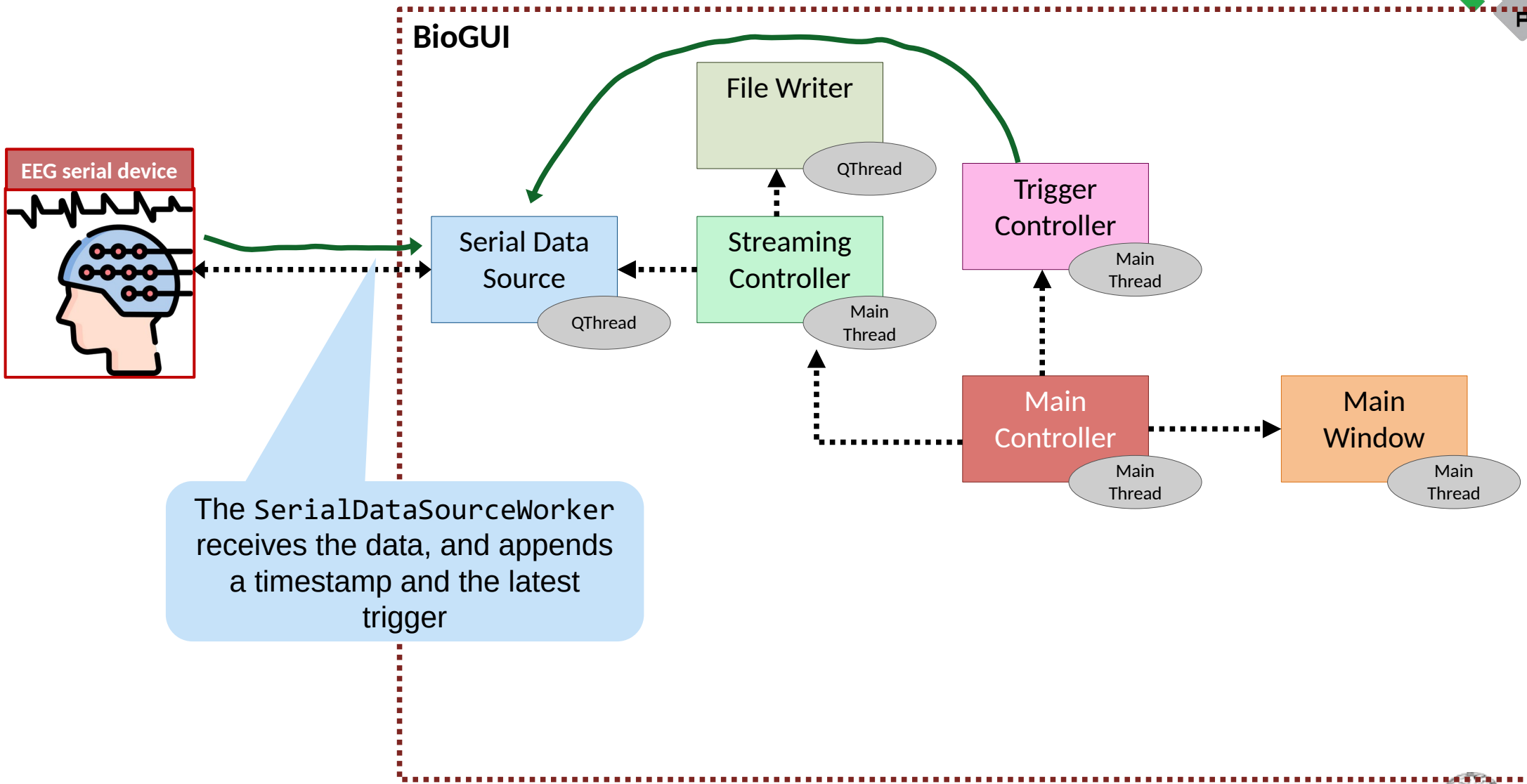
BioGUI Flow


BioGUI
File Writer
QThread
EEG serial device
Serial Data Source
QThread
Streaming Controller
Main Thread
Trigger Controller
Main Thread
User starts acquisition
Main Controller
Main Thread
Main Window
Main Thread

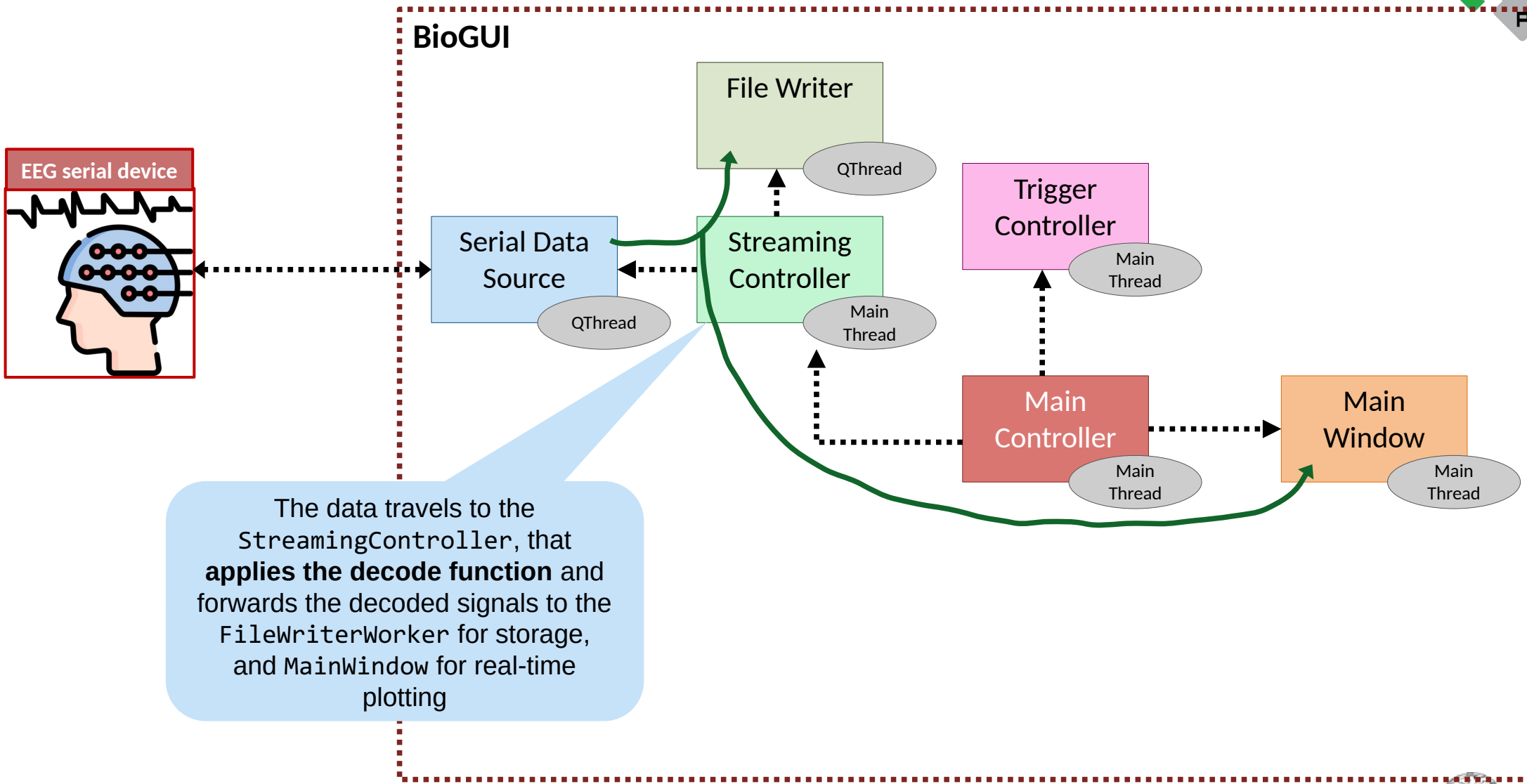
BioGUI Flow


BioGUI
File Writer
QThread
Trigger Controller
Main Thread
EEG serial device
Serial Data Source
QThread
Streaming Controller
Main Thread
Main Controller
Main Thread
Main Window
Main Thread
The start command travels to the DataSourceWorker via StreamingController

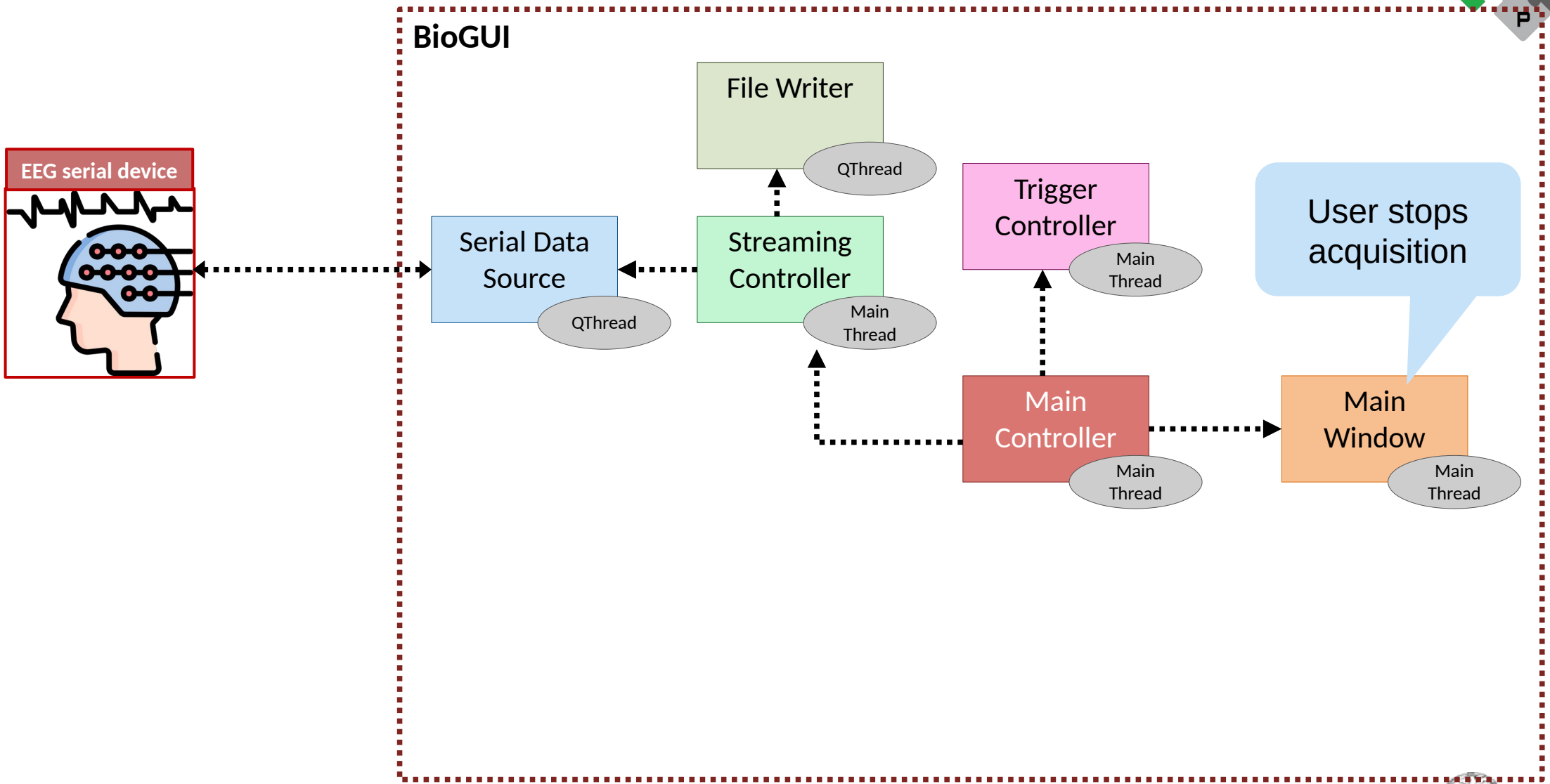
BioGUI Flow


BioGUI
File Writer
QThread
Trigger Controller
Main Thread
EEG serial device
Serial Data Source
QThread
Streaming Controller
Main Thread
Main Controller
Main Thread
Main Window
Main Thread
The SerialDataSourceWorker receives the data, and appends a timestamp and the latest trigger

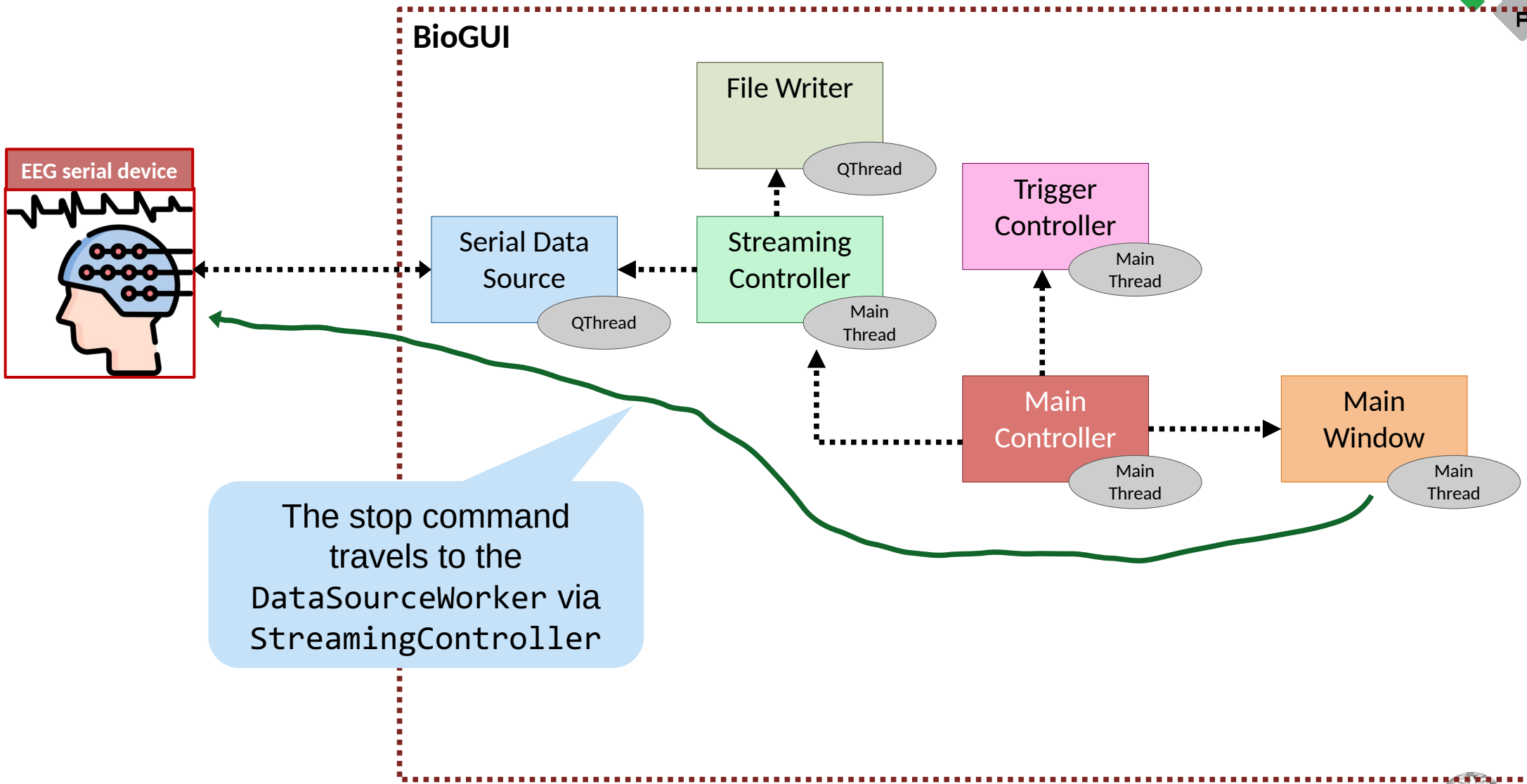
BioGUI Flow


BioGUI
File Writer
QThread
EEG serial device
Serial Data Source
QThread
Streaming Controller
Main Thread
Trigger Controller
Main Thread
Main Controller
Main Thread
Main Window
Main Thread
The data travels to the StreamingController, that applies the decode function and forwards the decoded signals to the FileWriterWorker for storage, and MainWindow for real-time plotting

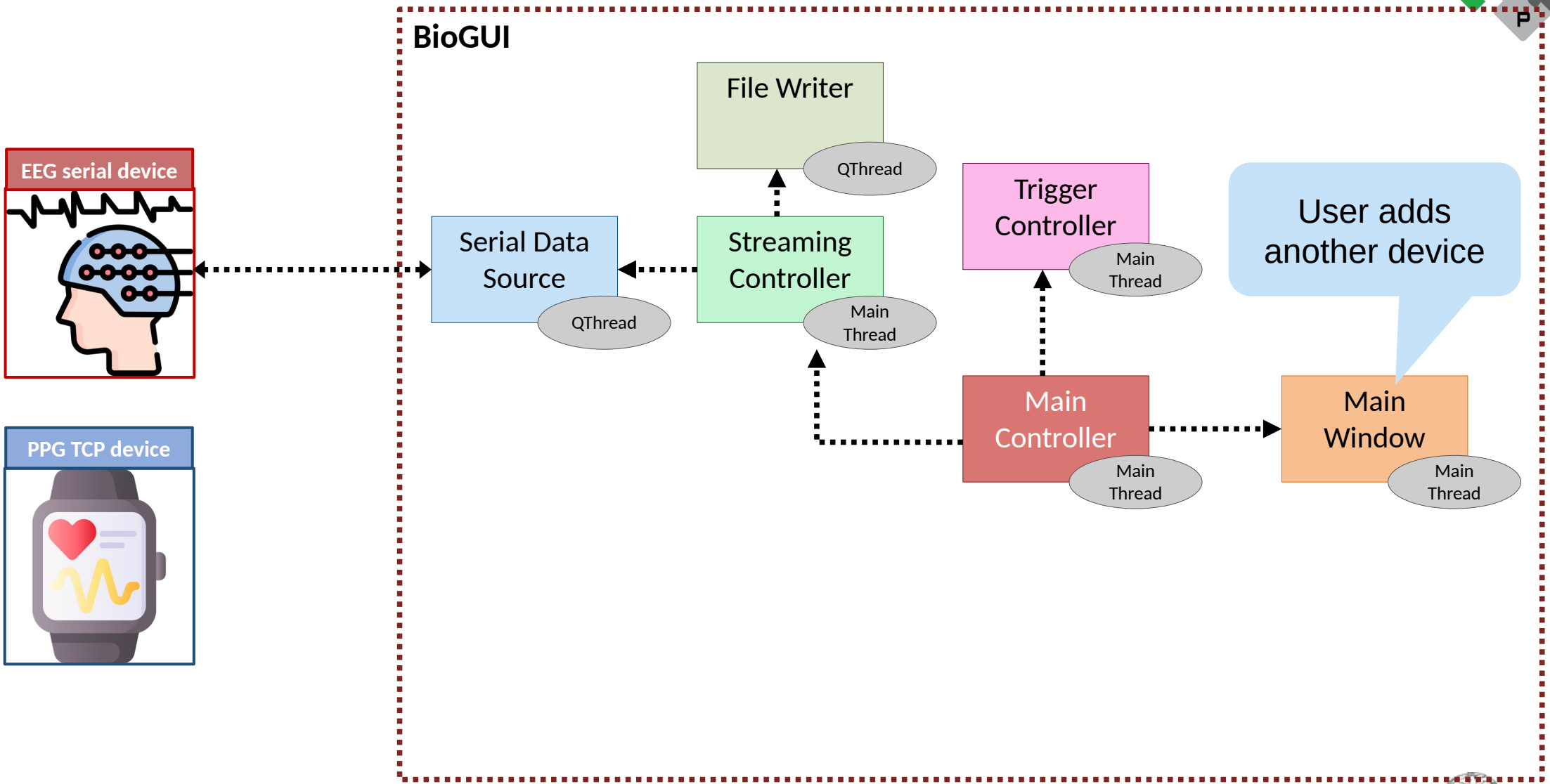
BioGUI Flow


BioGUI
File Writer
QThread
EEG serial device
Serial Data Source
QThread
Streaming Controller
Main Thread
Trigger Controller
Main Thread
User stops acquisition
Main Controller
Main Thread
Main Window
Main Thread

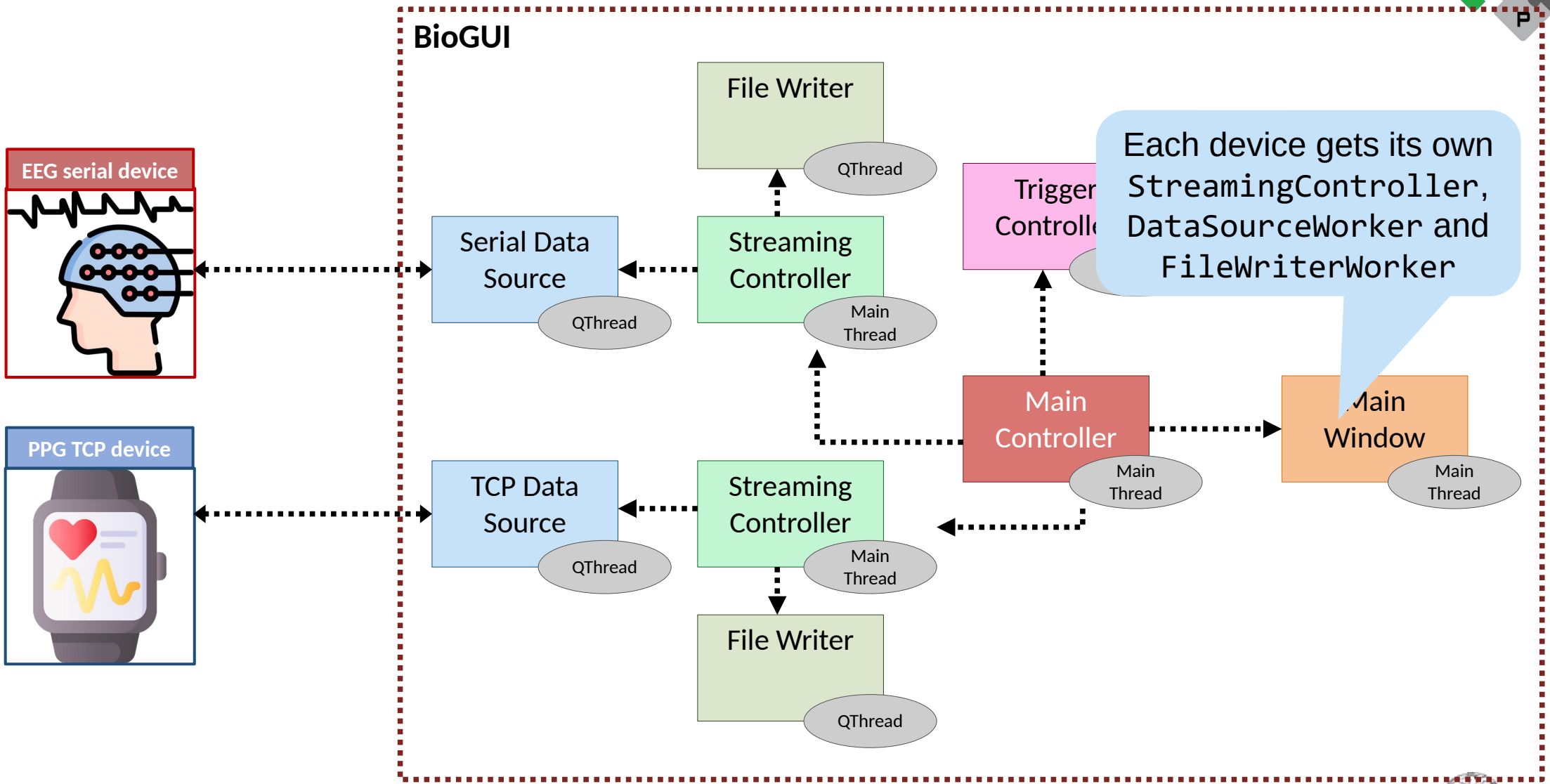
BioGUI Flow


BioGUI
File Writer
QThread
Trigger Controller
Main Thread
EEG serial device
Serial Data Source
QThread
Streaming Controller
Main Thread
Main Controller
Main Thread
Main Window
Main Thread
The stop command travels to the DataSourceWorker via StreamingController

BioGUI Flow


BioGUI
File Writer
QThread
Trigger Controller
Main Thread
User adds another device
EEG serial device
Serial Data Source
QThread
Streaming Controller
Main Thread
PPG TCP device
Main Controller
Main Thread
Main Window
Main Thread

BioGUI Flow

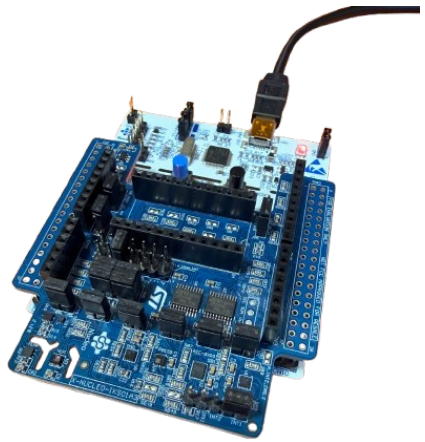

BioGUI
File Writer
QThread
Trigger Controller
Each device gets its own StreamingController, DataSourceWorker and FileWriterWorker
EEG serial device
Serial Data Source
QThread
Streaming Controller
Main Thread
Main Controller
Main Thread
Main Window
Main Thread
PPG TCP device
TCP Data Source
QThread
Streaming Controller
Main Thread
File Writer
QThread

Live demo

- **Connect the OTBioelettronica Sessantaquattro+**
- **Stream HD-sEMG and IMU data**

Hands-On

- **We'll interface the BioGUI with a STM32F401RET6 NUCLEO board, paired with a X-NUCLEO-IKS01A2 sensor expansion board**
  - 3-axis accelerometer
  - Gyroscope
- **Complete the Python interface file to enable communication**
- **Acquire a dataset using the trigger module**
- **Train a SVM**
- **Perform a control task via the forwarding module**
- **The code is available at github.com/pulp-bio/biogui_getting_started**



PULP Platform
Open Source Hardware, the way it should be!

Mattia Orlandi mattia.orlandi@unibo.it

Institut für Integrierte Systeme – ETH Zürich
Gloriastrasse 35
Zürich, Switzerland

DEI – Università di Bologna
Viale del Risorgimento 2
Bologna, Italy

ETH zürich ALMA MATER STUDIORUM UNIVERSITÀ DI BOLOGNA

@pulp_platform
pulp-platform.org
youtube.com/pulp_platform