CockpitController HID Handler

Overview

This project provides a robust, cross-platform HID handler for interfacing a DCS-BIOS-enabled ESP32-based cockpit device with DCS World (Digital Combat Simulator). It bridges the UDP multicast data stream generated by DCS-BIOS and a USB HID device (the ESP32), forwarding commands and state changes between DCS and the cockpit hardware.

The code is fully modular and architected for easy migration to C++ or other environments, with all multithreading, synchronization, HID protocol, and event loop logic clearly separated and documented.

System Architecture

Main Functional Blocks

1. **UDP RX Thread (`NetworkManager`)**
    - Listens to DCS-BIOS UDP multicast group.
    - Receives binary DCS-BIOS packets.
    - Chunks and encapsulates packets as HID OUT reports.
    - Forwards each chunk to every handshaked USB HID device via `dev.write()` (OUT report, Report ID 0).
    - Updates global statistics (frames, bytes, bandwidth).
2. **Per-Device Thread (`device_reader` in `hid_device.py`)**
    - Handles handshake with the device via HID FEATURE report (`send_feature_report()` for "DCSBIOS-HANDSHAKE").
    - After handshake, continuously reads device state changes (button presses, analog axes, etc.) via `dev.read()` (input report).
    - On valid input, fetches string data using `dev.get_feature_report()`, decodes as ASCII, and sends to DCS as a UDP unicast packet via the reply address detected from DCS UDP stream.
    - Handles device disconnect, reconnect, and resource cleanup.
3. **GUI Thread (`CockpitGUI`)**
    - Displays live device list, statistics, logs, and status.
    - Updates stats and event log asynchronously via a thread-safe queue (`uiq`).
    - Responds to events and updates user interface accordingly.

Threading Model

- **UDP RX Thread:**
  Single thread created by `NetworkManager` (`_udp_rx_processor`).
  Responsible for all DCS-BIOS UDP reception and forwarding to HID.
- **Per-Device Threads:**
  Each device gets its own thread via `device_reader_wrapper` in `CockpitGUI`.
  Responsible for handshake, draining input, sending device-originated messages back to DCS via UDP.
- **GUI/Main Thread:**
  Tkinter event loop, all UI activity, event queue processing, and stat display.
- **Thread Safety:**
  All shared data (`stats`, `reply_addr`, etc.) is protected via a global lock (`global_stats_lock`), and all cross-thread communication is performed via Python `queue.Queue` (for events/logs).

USB HID Protocol

HID Report Types

- **OUT Report (`write()`):**
  Used for forwarding DCS-BIOS UDP data to the ESP32.
  - Report ID: 0 (first byte)
  - Payload: Up to 64 bytes of raw DCS data
  - Length: Always padded to 65 bytes
- **FEATURE Report (`send_feature_report()`/`get_feature_report()`):**
  Used only for initial handshake ("DCSBIOS-HANDSHAKE") and fetching device string responses for input events.
  - Handshake: Sends "DCSBIOS-HANDSHAKE" (padded) as a FEATURE report
  - Input fetch: Reads FEATURE report for string data (e.g., button label and state)
- **IN Report (`read()`):**
  Used by device threads to read input state changes (button/axis events).

Firmware Expectations

- The ESP32 firmware must have an OUT endpoint and implement `_onOutput()` to receive DCS-BIOS data.
- FEATURE endpoint is only for handshake or string fetch (not main DCS data stream).
- The first byte of every HID report is the Report ID (`0`).

Data Flow

1. **DCS→UDP→Python→HID OUT→ESP32:**

    - DCS-BIOS emits UDP packets (multicast group).
    - `NetworkManager` receives packets, splits into 64-byte chunks.
    - Each chunk is wrapped as `[Report ID 0][chunk]` and padded to 65 bytes.
    - Each report is sent via `entry.dev.write(report)` to each connected, handshaked device.
2. **ESP32→HID IN/FEATURE→Python→UDP→DCS:**
    - ESP32 device generates input reports on state change.
    - Per-device thread detects and reads the report via `dev.read()`.
    - Calls `get_feature_report()` to fetch the actual ASCII-encoded command/label.
    - Forwards message to DCS as a UDP unicast packet to last detected sender (`reply_addr`).

Handshake Protocol

- On device connection, per-device thread attempts handshake by sending "DCSBIOS-HANDSHAKE" as a FEATURE report.
- Device responds with a matching FEATURE report.
- Only after handshake completes does the device participate in data forwarding.
- Disconnect and reconnection are detected and handled automatically.

Statistics and Event Logging

- All network and device activity is logged to an event queue (`uiq`), which the GUI thread displays.
- Stats (frames, bandwidth) are updated globally using a shared dictionary and lock for real-time display.
- Stats are reset and recomputed on a 1-second timer.

Synchronization and Thread Safety

- All communication between threads (GUI, network, device) is done via thread-safe queues and explicit locks.
- All mutable shared state (e.g., `reply_addr`, `stats`) is always accessed under `global_stats_lock` to avoid race conditions.

Configuration

- VID/PID and other settings are loaded from `settings.ini` at startup.
- All code paths (USB, network, GUI) read config from `config.py`.

Porting to C++/Qt/Other Platforms

- All threading is explicit and separable.
- All global/shared state is isolated (can be replaced with class members or static objects).
- All network/HID/event logic is modular, making it trivial to port to Boost.Asio, libusb, or Qt's networking and USB classes.
- Code flow matches standard producer/consumer or event-loop patterns.

Design Principles

- **Single Responsibility:** Each module and thread has exactly one job.
- **No Data Races:** Only one thread writes HID OUT, only one thread reads HID IN per device.
- **Explicit Protocol:** HID report formats, endpoint expectations, and buffer sizes are always documented and enforced.
- **No "Magic":** All cross-thread activity is logged and visible in code.
- **Future Proof:** All logic is ready for C++ migration by swapping threading, lock, and USB/HID APIs.

Troubleshooting

- If HID OUT reports are not received, check that the VID/PID matches and only one writer is used.
- If UDP frames do not increment, verify UDP multicast join and port in `settings.ini`.
- If stats or logs don't update, check that all shared dicts use the global lock.
- If events are missing, check event queue (`uiq`) and threading startup.

File Structure

- `config.py` — All configuration, stats, and shared globals
- `gui.py` — GUI logic, device list, stats/event display
- `network.py` — UDP RX/TX, frame forwarding, stat updates
- `hid_device.py` — Device enumeration, handshake, per-device event/logic

- `main.py` — Application startup and mainloop
- `settings.ini` — VID/PID and other runtime settings

Migration Guide for C++/Qt

- Replace Python `threading.Thread` with `std::thread` or `QThread`
- Replace `queue.Queue` with `std::queue` + mutex/condition variable or `QQueue`
- Replace Python sockets with `boost::asio` or `QUdpSocket`
- Replace `hidapi` calls with `libusb` or platform HID class (see descriptors for OUT report parity)
- Replace shared dicts/locks with atomic or mutex-protected structs/classes
- Main event loop can be translated directly to a Qt event loop or custom event dispatcher