CockpitController HID Handler – Python-to-C++ Migration Plan

1. Introduction

This document outlines a comprehensive migration plan to convert the **CockpitController HID Handler** (currently a Python application) into a high-performance, aerospace-grade C++ application. The target platform is 64-bit Windows 10/11, and the new implementation must meet strict real-time and reliability requirements. We describe the system's purpose, design decisions, and the architectural approach for the C++ solution. All aspects – from GUI design to HID and network communication, memory management, threading, and error handling – are addressed in detail. The goal is to ensure **deterministic behavior, near-zero idle CPU usage, and robust fault containment** in the final C++ application, adhering to the highest standards of safety and performance.

2. System Requirements and Goals

This section enumerates the strict requirements for the new C++ implementation, expanding on each to guide design decisions:

- **Target Platform – Windows 10/11 x64:** The application must be developed with Visual Studio for 64-bit Windows, ensuring compatibility with modern Windows OS features and drivers. All code and libraries will be 64-bit compliant.
- **Memory Management – Zero Heap Allocation:** Dynamic memory (heap) allocations at runtime are to be **avoided unless absolutely unavoidable**. The design will emphasize static allocation (global or file-scope static buffers) and stack allocation for local data. This avoids fragmentation and unpredictable latencies due to allocation/deallocation. Using static memory is a known best practice in real-time and embedded systems to avoid runtime delaysmedium.comembeddedartistry.com. Any necessary dynamic allocation (if needed for initialization or third-party libraries) will be performed only during startup, not during steady-state operation, to ensure consistent timing.
- **Exception Handling – No Runtime Exceptions:** C++ exceptions will not be used. The application will be compiled with exceptions disabled and Run-Time Type Info (RTTI) turned off. In Visual Studio, we will set "Enable C++ Exceptions" to **No** and use the `/GR-` compiler flag to disable RTTIstackoverflow.com. This ensures that no exception unwinding code is generated, reducing both binary size and runtime overhead. Error handling will be done via return codes or status values, following embedded C++ best practices (many safety-critical standards prohibit exceptions for predictability). The build will also use structured error handling for any OS calls if needed, rather than C++ exceptions.
- **Minimalistic Native GUI:** The current Python application's minimal GUI will be reimplemented using native Windows UI frameworks. We prefer the Win32 API directly for minimal overhead and full control, or a lightweight C++ GUI toolkit that can be

statically linked (e.g., Windows Template Library - WTL). The GUI will be a thin interface to display status and allow user interaction (start/stop, etc.), without heavy frameworks (no MFC/.NET). Using raw Win32 ensures no additional runtime dependencies and a small footprint. We will statically link the C runtime (`/MT` flag in MSVC for static CRT) so that the executable can run standalonestackoverflow.com. The GUI should closely replicate the Python version's look and feel (simple dialog or window with status text), and will utilize standard Win32 controls (e.g., `Static` text controls, `Button` controls) and a classic message loop (`GetMessage/TranslateMessage/DispatchMessage`). The use of `WIN32_LEAN_AND_MEAN` and targeted `<windows.h>` includes will minimize overheadlearn.microsoft.com.

- **Threading Model – Deterministic and Efficient:** Concurrency will be used to isolate tasks (HID handling, network I/O, GUI) into separate threads, improving determinism by dedicating threads to specific real-time tasks. We will use either C++11 `<thread>` or the native Win32 `CreateThread` API as appropriate. **Determinism and priority control are paramount:** using the Win32 API allows setting thread affinity and real-time priority levels directly. We will likely create dedicated threads for:
  - HID communication (high priority thread to service device I/O with minimal jitter),
  - Network I/O (another high-priority thread for UDP send/receive),
  - The GUI/main thread (normal priority for user interface).
    Each thread's priority and possibly CPU affinity will be tuned. For instance, we can assign the process to `REALTIME_PRIORITY_CLASS` and set critical threads to real-time range priorities (above 15)learn.microsoft.com, ensuring the OS scheduler minimizes their latency. However, care will be taken since Windows real-time priority can starve other processessuperuser.com. At a minimum, the HID and network threads will run at **HIGH** or real-time priority class, with the GUI thread at normal priority, to ensure time-critical I/O is never delayed by UI activity.
- **HID Communication – Precise Replication of Python Logic:** The C++ application's HID layer must **exactly replicate the behavior of the original Python code** in handling HID Feature reports, Output reports, and Input reports. This includes report formats, IDs, and timing. We will use the most efficient native method available:
  - Likely leveraging the Windows HID API (`hid.dll`) via functions like `HidD_SetFeature`, `HidD_GetFeature` for feature reports, and `HidD_SetOutputReport` or overlapped `ReadFile/WriteFile` on a device handle for input/output reports. The `HidD_SetFeature` routine, for example, is used to send a feature report to the HID devicelearn.microsoft.com. These functions operate at a low level with minimal overhead. We will include `<hidsdi.h>` and link against `Hid.lib` (provided by the Windows Driver Kit) to access these HID functions.
  - If appropriate, we may use a small cross-platform HID library like **hidapi** (statically compiled) for convenience. HIDAPI internally uses the Windows HID stack (SetupAPI + HidD_ functions) and provides `hid_read()`, `hid_write()`, and `hid_send_feature_report()` which align well with our needs. However, using it means introducing a third-party component; we will evaluate if its

overhead (dynamic allocations or not) meets our zero-heap goal. If not, a direct implementation using `CreateFile` on the HID device path and `WriteFile/ReadFile` with overlapped I/O will be written.

    - The HID module will enumerate and open the target HID device by its Vendor ID/Product ID (and interface usage, if needed) as specified in the configuration. Using the Windows SetupAPI, we can find the device path for the known VID/PID, then `CreateFile` to get a handle with `FILE_FLAG_OVERLAPPED` for async operations. All interaction with the device will mirror the Python logic (e.g., if the Python code sent certain byte sequences or poll rates, the C++ must do the same). Timing and ordering of feature vs output reports will also be preserved.
    - The HID thread will use **overlapped I/O** or blocking waits to achieve **near-zero CPU usage** when idle. For example, it can issue a `ReadFile` on the HID handle with an overlapped structure and wait on an event or use `WaitForSingleObject` on the device handle to be signaled when data arrives. This way, when no HID data is incoming, the thread sleeps and uses no CPU. No polling loops with timeouts will be used (unlike a naive Sleep-based poll) to keep idle CPU at essentially 0%.

- **Networking – UDP Multicast & Unicast, Non-blocking:** The Python application's network communication (likely broadcasting telemetry or receiving commands) will be reimplemented using Winsock2 for UDP. We must support both multicast (for receiving or sending on a group address) and unicast. The design will use **asynchronous socket I/O** so that no thread is permanently blocked and CPU usage is minimized:
    - We will create UDP sockets with `WSASocket()` specifying `WSA_FLAG_OVERLAPPED` for overlapped I/O. For multicast reception, the socket will be bound to the appropriate interface and use `setsockopt()` with `IP_ADD_MEMBERSHIP` to join the multicast group (with options for loopback control if needed).
    - **Overlapped I/O with Event or IOCP:** To avoid any blocking `recvfrom` calls, we will post asynchronous receive requests. One approach is using overlapped I/O with an event object:
        - Call `WSARecvFrom()` with an `OVERLAPPED` structure and an event handle. If data is not immediately available, Winsock will return `WSA_IO_PENDING` and signal the event when a datagram arriveslearn.microsoft.comlearn.microsoft.com. The network thread can then wait on this event (or multiple events) using `WSAWaitForMultipleEvents` without consuming CPU time. When signaled, it retrieves the data with `WSAGetOverlappedResult` and processes it, then immediately reposts another WSARecvFrom for the next packet. This technique provides continuous, non-blocking reception with efficient use of a single thread.
        - Alternatively, for higher scalability, we could attach the socket to an I/O Completion Port (IOCP) and have a dedicated IOCP worker thread to handle completions. Given our application likely has a small number of sockets (one for multicast, one for unicast), a simpler event-wait loop may suffice. IOCP is more beneficial when managing many sockets, but it also offers excellent efficiency with **zero polling**. If used, we would create an

    IOCP via `CreateIoCompletionPort`, associate both sockets, and use `GetQueuedCompletionStatus` in a loop to handle completed recv/send operations. This ensures even if multiple packets arrive quickly, the thread can handle them back-to-back.
  - **Non-blocking send:** Outgoing UDP messages (e.g., unicast commands or telemetry responses) will use `sendto()` on the UDP socket. UDP sends are typically non-blocking by default (they copy to kernel buffer and return immediately if the buffer is not full). If using overlapped, we might also use `WSASendTo` with overlapped for symmetry, but it's usually not necessary unless we want to queue multiple sends without waiting. In either case, sending will be designed to never stall the calling thread; if needed, we will check for `WSAEWOULDBLOCK` and either drop or buffer the packet (bounded buffer) rather than wait.
  - **Multicast/Unicast Handling:** The network module will parse incoming messages exactly as the Python version does. This includes handling any specific packet formats or protocols (for example, if the Python app is part of a simulation interface like DCS-BIOS, it might listen to multicast state updates and send unicast control messages). All packet structures (byte ordering, frequency, etc.) will be duplicated bit-for-bit. We will maintain any timing behavior (e.g., if Python throttled certain messages or coalesced them, the C++ should as well).
  - **No Busy Waiting:** By using overlapped I/O and event-driven waits, the network thread will consume **virtually 0% CPU while waiting** for data. This meets the *"near-zero idle CPU usage"* goal. The only CPU usage will be when processing a packet, which will be very short (parsing a small buffer and perhaps updating some state or forwarding it). Similarly, sending via `sendto` is quick. Thus, the network thread's design yields bounded response times to incoming data without wasteful loops.
- **Configuration via INI File:** The application will continue to use a `settings.ini` configuration file (same format as the Python version) for user-specific settings (device identifiers, IP addresses/ports, any tuning parameters). We will use the **inih** library or a custom INI parser that does not perform dynamic allocation. *inih* (INI Not Invented Here) is a simple, two-file library in C designed for embedded systems; it parses an INI file with minimal overhead and can be configured to use only stack memorygithub.com. We will integrate inih in static mode (it's just `ini.c` and `ini.h`) and call `ini_parse()` on startup to load the config. The parser will populate a global `Settings` struct with all needed fields. Since inih by default uses a fixed-size line buffer on the stack (and can be tuned via `INI_MAX_LINE`), it aligns with our no-heap policygithub.com. If for any reason inih is not suitable, we will write a simple parser: read the file line by line (using `ifstream` or `CreateFile`/`ReadFile` with a static buffer) and parse `key=value` pairs, which is straightforward given the small size of typical INI files. The configuration module will expose a read-only interface to other parts (e.g., `Settings::Get("udp_port")`) but the values will be stored in static or file-scope structures to avoid global new/delete.
- **Logging and Statistics – Fixed-size Buffers:** Logging and runtime statistics gathering will be implemented with zero dynamic growth. To avoid unbounded memory use, we will employ **circular buffers** (ring buffers) or fixed-size arrays for logs. For example, we

might allocate a static array of log message structures (with a max count) and always write new log entries in a wrap-around fashion, overwriting the oldest when full. Circular buffers are ideal for this purpose as they use fixed memory and simply wrap pointers as new data arrivesembeddedartistry.comembeddedartistry.com. Each module (HID, network, etc.) can have its own ring buffer for event logging. The GUI can periodically display or dump these logs (for example, showing the last N log lines in a multiline text control). Likewise, any statistics (counters of packets, error counts, timing measurements) will be stored in predetermined structures of fixed size. We will **not** use resizable containers like `std::vector` or `std::string` for logs in normal operation; if strings are needed, we will use fixed char buffers or pre-allocated pools. The logging mechanism will be carefully designed to avoid excessive overhead on real-time threads: possibly using lock-free enqueue or at least minimal locking (e.g., using an atomic index to produce/consume logs). The logging should never block a high-priority thread; if the buffer is full, we will either drop the new message or overwrite oldest (as per design choice) rather than block or allocate more memory. This ensures **bounded memory usage** and consistent performance.

- **Protocol Fidelity:** All application-level protocols and data formats will remain consistent with the Python version. Any specific packet framing, checksums, sequence of HID operations, etc., will be duplicated. Before implementation, the Python code will be reviewed to extract these details. For instance, if the Python app expects certain 16-bit fields or sends a feature report to trigger a mode in the device, the C++ must do the same. This may include byte-order (endianness) considerations and timing between messages. The migration aims for a **drop-in replacement** from a functional perspective – external behavior should be unchanged, aside from performance improvements.
- **Performance and Reliability Goals:** The final design must ensure:
  - **Near-Zero Idle CPU**: When the system is not actively processing HID or network events, CPU usage should drop to effectively zero. This is achieved by using blocking waits or events (as described) instead of polling loops. Each thread will sleep when there's nothing to do.
  - **Deterministic Timing (Bounded Latencies)**: The system should react to inputs within a predictable maximum time. By using dedicated threads with high priority, and avoiding unpredictabilities like garbage collection (in Python) or memory allocation, we reduce jitter. For example, a HID report arriving will be picked up by the HID thread almost immediately (thread awakened by the OS I/O completion), and processed without delay. Communication between threads (if any) will use lock-free queues or short critical sections to ensure minimal waiting. We also plan to leverage real-time priorities and possibly isolate threads to specific CPU cores if needed for further determinismlearn.microsoft.com. All critical code paths will be measured to ensure they complete quickly (likely microseconds to a few milliseconds worst-case, easily within typical real-time simulation frame requirements).
  - **Clean Shutdown**: The application must close cleanly without resource leaks, zombie threads, or dangling handles. We will implement a controlled shutdown sequence: e.g., on exiting the GUI or receiving a quit command, signal all threads to terminate (using an atomic flag or event), wake them if needed (by posting dummy I/O), wait for threads to join, close device handles, sockets, and then exit.

Each thread will regularly check a shared "shutdown requested" flag so it can break out of its loop promptly. All handles (device handles, socket descriptors, event objects) will be closed in the proper order. By avoiding dynamic memory, we also minimize leak risks, but we will still pair any rare allocations with proper deletion at program end. The result is no lingering processes or threads once the app exits.

- **Fault Containment**: Each module will handle errors internally and **not crash the entire application** if possible. For instance, if the HID device is unplugged or a read fails, the HID thread will catch that error (via a returned error code from `ReadFile` or a timeout) and will log it and attempt recovery (e.g., try to reinitialize the device periodically) or at least signal the GUI to show a warning. It will not throw exceptions (none used) or segfault. Similarly, if the network socket experiences an error (like the network cable unplugged or a send fails), the network thread will handle the `WSAGetLastError()` code, log the event, and perhaps attempt to re-bind or wait until the network is back, rather than crashing. Each thread is largely independent – an issue in one (e.g., network down) does not directly corrupt the state of others thanks to clear module boundaries. This "containment" strategy is aligned with robust aerospace software design: fail-safe and fault-tolerant operation.
- **C++ Code Safety and Style**: The code will comply with stringent C++ safety and style standards, analogous to MISRA C++ or JSF++ guidelines used in aerospace. This means: use of **RAII** (where applicable) for resource management (even though exceptions are off, RAII still ensures deterministic release of resources on scope exit), avoidance of dangerous casts or pointer arithmetic, careful use of `volatile` or atomics for shared data, and following const-correctness. We will enable high warning levels (`/W4` or `/Wall`) and treat warnings as errors (`/WX`) to enforce code quality. Static analysis tools or Visual Studio's Code Analysis will be used to catch any potential issues like buffer overruns or race conditions. The coding style will emphasize clarity and maintainability: for example, using explicit structures for packets rather than raw byte arrays when possible, and adding comments for any tricky low-level code. The design favors straightforward, **predictable constructs over fancy meta-programming**, given that safety and clarity are priority.

Having established the requirements, the next sections describe the system architecture and how each module is designed to fulfill these goals.

3. High-Level Architecture and Design Decisions

3.1 Overall Architecture and Module Breakdown

The application is structured into modular components, each corresponding to a major function (mirroring the Python application's responsibilities). The modules and their primary responsibilities are:

- **Main Application / Core**: Orchestrates initialization and shutdown. It loads configuration, starts all modules (spawning threads for HID and network), and initializes the GUI. It also coordinates inter-module communication where necessary (for example, passing user commands from GUI to the other modules). This core will ensure all threads are created with the correct priority and are monitored during execution.
- **HID Module**: Handles all communication with the USB HID device (cockpit controller hardware). This module encapsulates device discovery (finding the HID by VID/PID), device opening/closing, and all input/output exchanges. It runs in its own thread (HID Thread) at high priority. Internally, it will implement the logic for reading input reports from the device (e.g., button presses, switch toggles) and sending output or feature reports to the device (e.g., LED updates or configuration commands). The HID module provides an interface to other parts of the app such as:
  - `HIDModule::SendOutputReport(data)` – queue or send an output report to device.
  - `HIDModule::SendFeatureReport(data)` – send a control feature report.
  - It might also callback or signal when an input event is received (e.g., a new button state), which the network module may need to transmit. This could be done via a thread-safe queue or direct call if thread-safe.
- **Network Module**: Manages UDP networking. Runs in its own thread (Network Thread), also high priority. It opens the necessary sockets (per config: e.g., one for listening to a multicast group from the simulation, and one for sending unicast commands or vice versa). It is responsible for encoding and decoding the application-level protocol. For example, on receiving a UDP packet, it parses it and updates relevant state or triggers a HID action (e.g., if a command to change an output came from the network, it will call into HID module to update the device). Conversely, if the HID module reports a local hardware event (e.g., the user flipped a switch), the network module will package that into a UDP message and send it out. The module ensures all socket I/O is asynchronous and non-blocking as discussed. The interface from this module might include:
  - `NetworkModule::SendPacket(buffer, dest)` – to send a prepared message (for use by others if needed).
  - Internally, it might have a listener that calls a handler function for each received message type, which could interact with HID or other modules.
- **GUI Module**: Provides the native Windows GUI for the application. Typically, this will run on the main thread (Win32 GUI must run in a single GUI thread with a message loop). It creates a window (or dialog box) with basic controls (for example: **Status indicators** for device and network connectivity, perhaps a **Connect/Disconnect** or **Start/Stop** button, and maybe a text area to display recent log messages or statistics). The GUI module communicates with the core and other modules via thread-safe mechanisms:
  - It can post commands (using `PostThreadMessage` or simpler, set some atomic flags) to signal threads. For instance, if the user clicks “Exit” or “Connect”, the GUI will signal the core to cleanly shut down or (re)initialize the HID/network.
  - It can periodically query status from other modules (possibly the core could aggregate some status info in a safe way that the GUI thread can read).
  - The GUI will also display data such as a running count of packets sent/received, or any warnings (e.g., “Device disconnected”).

    The GUI module will be implemented with Win32 calls (e.g., `CreateWindowEx` for window and controls). We will avoid any heavy GUI frameworks to keep within static linking and performance requirements. All rendering will rely on GDI for text (which is sufficient for a simple interface).

- **Configuration Module**: A utility component responsible for reading and storing configuration values from `settings.ini`. This is not a separate thread; it runs during initialization. It provides an easy lookup for settings throughout the app (e.g., `Config::UDPGroupAddress()` or `Config::DevicePID()`). This module will likely use a simple struct or class to hold values and possibly a small function to parse the INI using inih as described. Because configuration is only read at startup (and maybe saved at shutdown if needed), it does not impact real-time behavior and can safely use file I/O and parsing at init time.
- **Logging/Stats Module**: This may be a singleton or simple set of functions accessible by all threads for recording events. It will manage the circular buffers for logs. Each module can log via something like `Logger::Log(module, level, message)`. The logger will timestamp the entries (likely using a high-resolution timer or `GetSystemTimePreciseAsFileTime` for time-of-day). Because we can't allocate strings, the logger might use a pre-allocated pool of message buffers; or we format into a static char buffer and copy into the ring. We also consider using lock-free techniques: e.g., a write index that each thread atomically increases when writing a log line to a global log buffer. This module will also maintain counters (e.g., total packets received, last packet timestamp, device read errors count, etc.) which can be displayed or used for health monitoring. All such data will be stored in fixed-size arrays or structures.

These modules interact as shown below:

```
sql
CopyEdit
               +------------------------+           Network I/O (UDP)
               |    Network Module      | <---------------------------------
               |  - UDP recv/send       |                ▲ (Multicast Rx)
               |  - Parse incoming     |                | (Unicast Tx)
               +------------------------+                 |
                   |                   ▲                 |
                   | Outgoing HID data |  Inbound ctrl   |
                   | (e.g., commands)  |    msgs         |
                   ▼                   |                 |
+---------+   uses    +------------------------+    uses  |
|  HID    |---------->|        Core           |<----------+
| Module  |   via     |  (Main/App coordination)         |
|(Device  |   core    |   - Threads startup/shutdown     |
|  I/O)   |          /|\ - Aggregates status             |
+---------+           |   - Error handling decisions     |
    |  ^              |                                  |
    |  | HID events   |          GUI commands            |
    |  | (e.g. switch)|                                  |
    v  | change       |                                  ▼
+------------------------+                         +--------------------+
|     GUI Module         |<-------------------------|  Logging/Stats     |
|  (Win32 Window,        |    display status/logs   |   Module           |
|   User Interaction)    |                         +--------------------+
+------------------------+
```

Figure: Module interaction overview. The HID and Network modules operate largely in parallel, communicating via the Core. The GUI presents information to the user and sends user actions to the Core. The Logging module collects events from all parts for debugging/analysis. Solid arrows indicate primary data flow: e.g., HID events go to Network (to be sent out), network commands go to HID (to update hardware). Dashed arrows indicate control or indirect flows (e.g., GUI pulling logs, or core instructing modules to start/stop).

Module Boundaries and Isolation: Each module is implemented in a separate C++ class or namespace, with clear interfaces. They communicate through well-defined channels (method calls, message queues, or shared buffers) rather than large shared global data. This encapsulation improves fault containment – if something goes wrong in one module, it does not directly corrupt another. For example, the HID module will have an internal thread loop and will not expose the raw device handle to other modules – any device operation needed by others is done via HID module's methods, which internally synchronize as needed. This enforces that **only the HID thread touches the device**, eliminating race conditions on the device file handle. Similarly, only the Network thread will manipulate socket APIs directly.

3.2 Data Flow and Thread Interaction

To guarantee determinism and avoid deadlocks, we carefully design how data flows between threads:

- **HID Input -> Network Output:** When the HID device sends an input report (e.g., a switch flipped), the HID thread receives it (via overlapped I/O completion). It will parse the report into a meaningful event or value (based on the device protocol). Then, this information needs to be sent to the simulation via UDP. We have two options:
    1. **Direct call into Network module:** The HID thread can call a thread-safe function of the Network module like `NetworkModule::SendPacket(data)`. Inside, that function would prepare a UDP packet and use `sendto()`. Sending on a socket from a different thread than the one receiving is generally safe in Winsock (sockets can be used from multiple threads). However, we must ensure thread safety for any shared data (like if Network module caches the destination address or maintains a sequence counter, those must be protected). This direct method has low overhead and minimal latency (HID event triggers immediate network send). We will use lightweight synchronization (like an atomic or a mutex around the send preparation if needed) to ensure no data races.
    2. **Message queue between HID and Network threads:** Alternatively, the HID thread could enqueue the event into a lock-free queue, and the Network thread wakes up to dequeue and send it. This adds a slight latency (the network thread might be sleeping on I/O, though we could wake it via an event), but it cleanly separates the threads' responsibilities (HID thread doesn't call Winsock directly). A single-producer, single-consumer circular queue could be used, which avoids locks and dynamic mem. Given performance requirements, either approach is acceptable; the direct call is simpler and likely fine if carefully managed.

We lean towards **direct invocation** to minimize latency – the code will be written to ensure it's safe (the Network send function will be made thread-safe). Because UDP send is quick and does not block the HID thread (worst case it queues in kernel), this should not disturb HID processing.

- **Network Input -> HID Output:** Conversely, when a UDP command comes in (e.g., instructing the hardware to change a setting or LED), the Network thread will parse it and then call into the HID module. Similar design choices apply:
  - The Network thread can directly call `HIDModule::SendOutputReport(data)` to update the device. If the HID module ensures thread-safe output (e.g., by using an internal mutex for writing to device or by simply performing the `HidD_SetFeature` or `WriteFile` call directly if it's quick), this is the fastest route. HID output reports over USB are typically very fast (just a few bytes control transfer), but if the HID thread is also reading, we need to ensure no conflict. One approach: **serialize all HID writes in the HID thread.** We might achieve this by having the HID thread also be responsible for sending outputs, not just reading. In that case, the Network thread would place the desired output data into a queue or set a variable, then signal the HID thread (via an event or just by the HID thread noticing new data) to perform the actual `WriteFile/HidD_SetOutputReport`. This way, all USB transactions (in and out) happen on one thread, avoiding any simultaneous access. This is slightly more complex but provides clear ownership.
  - However, USB HID drivers do allow concurrent read and write from different threads in user-mode as long as the calls are synchronized at driver level. To be safe and "aerospace-grade," we prefer not to rely on that implicitly. **We will implement output queuing in the HID module:** The Network module calling `HIDModule::SendOutputReport` will actually enqueue the request into a lock-free ring inside HID module, then signal the HID thread via an event or condition variable. The HID thread, which is anyway either waiting for input or in its loop, will wake up, see the request, and perform the actual device write. The added delay is minimal (order of milliseconds or less) and it guarantees no race on the HID device handle. This approach exemplifies *deterministic design*: each thread does its domain's I/O, communication between threads is done with fixed-size queues and signals, avoiding unpredictable locks.
- **GUI Interaction:** The GUI thread mainly interacts with the Core:
  - When the user clicks a button (say "Disconnect" or "Quit"), the GUI will send a message or call a function in the Core, which then sets the appropriate flags to stop threads or close resources. We might use `PostMessage` to a hidden window or use custom window messages to trigger actions.
  - The GUI may also poll for status to update indicators (e.g., a "Device Connected" LED icon in the UI). This can be done via a timer in the GUI thread that every second queries the Core/HID module status (maybe an atomic boolean "deviceIsConnected" that the HID thread updates). Since it's read-only in GUI, this is safe if atomic or properly protected.
  - For logging, the GUI might have a "Log Viewer" area. We won't update this in real-time for every log entry (that could flood the GUI). Instead, the GUI could

use a timer or a separate low-priority thread to periodically copy new log entries from the circular buffer to the screen (with proper synchronization). This ensures the high-priority threads are never blocked by GUI rendering.

- **Synchronization and Priority Inversion:** We must be cautious to avoid scenarios where a high-priority thread waits on a low-priority thread (classic priority inversion). Our design mostly prevents this by design:
  - HID and Network threads run largely independently and do not wait on GUI or each other (except maybe briefly on a queue that is usually empty). If the HID thread enqueues data for Network, it doesn't wait for acknowledgment, it just continues.
  - If we use a mutex for, say, protecting a common log structure, a high-priority thread could be momentarily blocked if the GUI (low priority) holds the lock. To avoid this, we will design log writing to be either lock-free or at least split into separate buffers per thread to eliminate contention. Alternatively, use try-lock and skip logging if busy (logging is non-critical).
  - Another measure: we can use the Windows API `SetPriorityClass` and `SetThreadPriority` to finely tune priorities. For example, give HID thread highest, network second highest, logger thread (if any) below, and GUI normal. Also, we might use `SetProcessPriorityClass(hProcess, HIGH_PRIORITY_CLASS)` (or realtime if truly needed and safe) so that Windows schedules our process favorablylearn.microsoft.com. However, we will refrain from REALTIME_PRIORITY_CLASS unless testing shows we need it, due to its hazards (it can starve system threads if misused).

In summary, the data flow is streamlined and **predictable**: each critical action involves at most a small handoff between two threads (HID->Network or Network->HID), using non-blocking mechanisms. This ensures bounded latency for end-to-end processing (e.g., a device input to network output might take under 1 ms in practice: tens of microseconds for the USB read completion, a few micro to package and send UDP, and then the network stack latency). All operations are either lock-free or use very short locking intervals to maintain real-time performance.

3.3 Memory Model and Deterministic Memory Usage

The memory model of the application is designed to fulfill the **zero-runtime-allocation** requirement and to ensure deterministic memory access patterns:

- **Static Allocation Strategy:** Global or static structures will be used for all major data pools. For example:
  - Preallocate a fixed-size array of log entries (e.g., 1000 log records).
  - Fixed-size buffers for network I/O (e.g., a receive buffer of maximum UDP packet size, perhaps 2048 bytes, allocated on stack or static for each socket).
  - HID report buffers: we know the HID report sizes from the device's HID descriptor (for instance, 32 bytes feature report, 8 bytes input report, etc.). We will allocate static byte arrays of those lengths for building and parsing reports.

    These buffers can be allocated once and reused for every transaction (e.g., the HID thread can reuse the same buffer each time it calls `ReadFile` for input).
  - The configuration values will be stored in fixed-size char arrays for strings (e.g., if the INI has a string for an IP address, we can cap at 15 characters + null for IPv4, etc.) or appropriate scalar types for numbers. The inih library by default uses stack for parsing lines and does not allocate heap unless we override thatgithub.com.
  - Any dynamic memory that is absolutely required (for example, if using std::thread, internally it may allocate a small block for thread object or the CRT may allocate for IO buffers) will be constrained to initialization. We will document any such cases and ensure they don't occur in steady-state loops.
- **Stack Usage:** Using the stack for temporary data is preferred for speed and automatic cleanup. Each thread will have its own stack (we can specify a size in the linker if needed to ensure enough space, default is usually plenty). For example, when a network packet is received, the overlapped WSARecvFrom writes into a stack buffer (provided in a `WSABUF`). Since the maximum size is known (worst-case, maybe a few KB), that's safe. The thread then processes it, perhaps copying a few values out into another struct, and then discards it. This way, memory usage is consistent and does not fragment.
- **Memory Pools (if needed):** If some part of the system inherently requires allocation (for instance, if we decided to use an STL container for convenience in a non-critical part), we would use a custom allocator that draws from a static memory pool. However, our intention is to avoid STL in real-time parts altogether (no `std::vector` or `std::string` in HID/Network threads). For any required string manipulation (like building a log message or formatting data), we will use `snprintf` into fixed char buffers. If complex data structures are needed (say, mapping device addresses to actions), we can use fixed arrays or `std::array` of fixed size, or perfect hashing at compile time. The small scale of our application (one device, a few message types) means we likely don't need any large dynamic data structures.
- **Preventing Fragmentation and Leaks:** By not using malloc/free at runtime, we eliminate fragmentation issues completely. Also, static allocation means the memory footprint is largely fixed and predictable (which is desirable in aerospace contexts). We will verify using tools or by instrumenting operators new/delete to ensure they are not called after init. The absence of heap usage not only improves determinism but also simplifies verification (memory usage can be reasoned about and tested easily).
- **Thread-Local Storage:** Each thread might have some thread-local buffers to avoid sharing. For example, the HID thread can have a static thread-local buffer for constructing reports. This avoids needing to lock or coordinate buffer usage between threads – each uses its own. Thread-local storage is cheap and allocated on thread start (on the heap or internally, but since threads are only created at init, it's acceptable). We will be mindful of the limited number of threads (just a few), so TLS or simply static objects inside the thread function would suffice.
- **DMA and Hardware Access:** (If applicable) The HID USB I/O goes through the OS and driver; we don't directly manage those buffers beyond providing user-mode buffers. Winsock for UDP likewise uses internal buffering. We trust the OS for these, but from our side, we ensure not to create bottlenecks. For example, if a burst of UDP packets arrives, our network thread will process one by one quickly. The Winsock buffer

(SO_RCVBUF) should be tuned if needed (we can setsockopt to enlarge it if expecting bursts). This prevents packet loss and thereby unpredictable behavior. Similarly, for HID, if the device sends data faster than we read, there's an internal HID driver buffer of typically one or a few reports; we ensure our thread is always waiting and reading promptly to not overflow it.
- **Memory Alignment and Cache:** We will consider aligning important buffers to cache lines (64 bytes) if needed, and grouping frequently used data to avoid cache thrashing. For instance, log structures might be padded to avoid false sharing if logs are written from multiple threads. These micro-optimizations support performance but will be used judiciously (the primary performance gains come from eliminating dynamic allocations and context switches, which we have done).

In summary, the memory model is **static, fixed, and predictable**. This approach, common in safety-critical software, provides the foundation for consistent performance and simplifies verification (we can calculate worst-case memory usage easily and ensure no risk of out-of-memory errors at runtime).

4. Detailed Module Design

This section details each module's implementation plan, including key classes, functions, data structures, and how they meet the requirements. Pseudo-code and relevant code patterns are provided to illustrate important aspects.

4.1 HID Interface Module Design

Responsibilities: Discover and connect to the HID cockpit controller device; perform all data exchanges (input, output, feature reports); translate raw HID reports to internal events and vice versa. Maintain device state (e.g., connection status).

Implementation Class: `class HIDModule` (singleton or managed by core). Key elements of this class:

- **Device Identification:** Hardcoded or configurable attributes for the HID device, e.g., USB Vendor ID and Product ID (from `settings.ini`). Possibly usage page and usage ID if needed to distinguish among HID interfaces. These will be read from the Config module at startup.
- **Device Handle Management:** `HANDLE deviceHandle;` stored in the class when open. Functions:
    - `bool OpenDevice()` – uses Windows SetupAPI to find the device. We call `SetupDiGetClassDevs` for GUID_DEVINTERFACE_HID (or use HidD_GetHidGuid), then `SetupDiEnumDeviceInterfaces` and `SetupDiGetDeviceInterfaceDetail` to get a device path for our VID/PID. Then call `CreateFile(devicePath, GENERIC_READ|GENERIC_WRITE, ... FILE_FLAG_OVERLAPPED, ...)` to obtain `deviceHandle`. If found, also call `HidD_GetPreparsedData` and `HidP_GetCaps` to get capability info like

    `InputReportLength` and `FeatureReportLength` (to know buffer sizes). On success, also prepare an `OVERLAPPED` struct and event for reading.
    - `void CloseDevice()` – closes the handle and frees any resources. To be called on shutdown or on device error.
- **Thread Function:** `static DWORD WINAPI HIDThreadFunc(LPVOID param)` (if using CreateThread) or the `operator()` for a std::thread. This is the main loop:

```
cpp
CopyEdit
HIDModule::HIDThreadFunc() {
    while (!shutdownRequested) {
        // Issue an asynchronous read for the next input report
        ReadFile(deviceHandle, inputBuffer, reportSize, NULL,
&overlapped);
        // Wait for either the read completion or a shutdown event
        HANDLE events[2] = { overlapped.hEvent, shutdownEvent };
        DWORD wait = WaitForMultipleObjects(2, events, FALSE,
INFINITE);
        if (wait == WAIT_OBJECT_0) {
            // Read completed
            DWORD bytesRead;
            if (GetOverlappedResult(deviceHandle, &overlapped,
&bytesRead, FALSE)) {
                HandleInputReport(inputBuffer, bytesRead);
            } else {
                DWORD err = GetLastError();
                if (err == ERROR_DEVICE_NOT_CONNECTED) {
                    LogError("HID device disconnected");
                    // attempt reconnect or break
                } else {
                    LogError("HID read error", err);
                }
            }
        } else if (wait == WAIT_OBJECT_0 + 1) {
            // Shutdown signaled
            CancelIo(deviceHandle);
            break;
        }
        // After handling input, loop to issue next ReadFile again
    }
}
```

  This pseudo-code outlines a pattern: Always keep an overlapped read pending (to capture device data as soon as it arrives) and use an event to sleep while waitinglearn.microsoft.com. On data arrival, `HandleInputReport` is invoked.

- **Input Processing:** `void HandleInputReport(BYTE* data, DWORD len)`. This function interprets the raw report. For example, if the device sends a report with button states or axis positions, we decode it. Then we translate that into an application event. E.g., "Button X pressed" or "Switch Y set to position 2". Likely we map these to specific simulation commands. The mapping logic might be simple (one-to-one) or involve lookup tables (which could be defined in config or code). The result will be passed to the Network module. Possibly:

```
cpp
CopyEdit
if (event == BUTTON_X_ON) {
    networkModule->SendSimCommand("BUTTON_X", 1);
}
```

or something similar, depending on protocol (if DCS-BIOS or custom, it might be a binary message instead). The key is that this function runs in HID thread context, so it should be short. We avoid heavy work here – just parse and hand off.

- **Output Handling:** The HIDModule provides methods to send outputs. Typically:
    - `bool SendFeatureReport(const BYTE* report, size_t length)`: wraps `HidD_SetFeature(deviceHandle, reportBuffer, length)`. Since `HidD_SetFeature` is synchronous (it sends a control transfer and waits for ack), we must ensure not to call it in a context that could block a real-time thread too long. Feature reports are usually rare (configuration commands), so calling directly from network thread might be fine. But to be consistent, we might funnel it through the HID thread as well. We could post a request and have HID thread call `HidD_SetFeature` (which might block a few ms). Alternatively, since feature reports might only be sent at init or config changes, doing it inline on network thread (set at high priority) could be acceptable.
    - `bool SendOutputReport(const BYTE* report, size_t length)`: This could be implemented via `WriteFile(deviceHandle, reportBuffer, length, ...)` if the device has an output interrupt endpoint. Or Windows provides `HidD_SetOutputReport` to send it as a control transferlearn.microsoft.com. We will check the device's capabilities. Many HID devices use the control pipe for output if they don't have a separate endpoint, in which case `HidD_SetOutputReport` is appropriate. It's similar to SetFeature but for output reports. This function would likewise likely be called as a result of a network message (e.g., simulator says "turn on light 5", we prepare an output report with that bit and send to device).
    - Internally, to avoid concurrency issues, these functions in HIDModule could simply enqueue the report data to an internal queue and set an event. The HID thread loop can be extended to wait for output events as well (or poll a queue each loop iteration). One design is to incorporate it into the same Wait as input:
        - We have two events: one for read complete, one for "output pending".
        - If output pending event is signaled (from another thread), the HID thread wakes, grabs the next output report from the queue, performs `WriteFile` or `HidD_SetOutputReport`, then goes back to waiting for either new output or new input.
        - This way, all writes happen in HID thread. The trade-off is slightly increased complexity, but it's manageable and aligns with our safety-first approach.

    - The output queue can be a small fixed ring (since outputs likely infrequent). If multiple network commands come in rapidly, we queue them and process sequentially, ensuring the device isn't flooded (HID can typically handle only one report at a time anyway).
- **Error Handling:** Within the HID thread, if any call fails:
    - On `CreateFile` failure (device not found), OpenDevice returns false. We then either retry after some delay or report to GUI and wait for user action. Possibly, the app can start in "disconnected" state and periodically attempt to connect if configured to do so.
    - On `ReadFile` overlapped indicating device removal (`ERROR_DEVICE_NOT_CONNECTED`), the thread should clean up the handle, notify other parts (set a status flag "device lost"), and attempt reconnection. Perhaps wait a few seconds then call `OpenDevice` again in a loop until it returns true or shutdown is signaled. Logging each attempt would be useful for debugging. This ensures resilience if the USB cable is unplugged mid-flight and replugged.
    - On `WriteFile`/HidD_SetOutputReport failure, similar approach: log the error and possibly treat it as a disconnect if persistent.
    - All errors are caught locally; none cause an exception up the stack. The thread can recover or at worst exit gracefully (notifying the core to maybe try restarting it). The rest of the app stays running (network could still send/receive perhaps, or GUI remains active to allow user to reconnect device).
- **Headers and Libraries:** For HID, we will include `<windows.h>`, `<winioctl.h>` (for device IOCTL definitions if needed), `<hidsdi.h>` (for HidD_* functions), and link against `hid.lib` (and `setupapi.lib` for device enumeration). We define `WIN32_LEAN_AND_MEAN` to exclude unnecessary Win32 components. We also ensure to include `<strsafe.h>` or use safe string functions for any buffer operations to avoid overflow.

Summary: The HIDModule is designed to handle device I/O in a self-contained thread, exposing a safe interface for others to request device actions. By using overlapped I/O and proper synchronization, it achieves **efficient blocking waits (no CPU use while idle)** and thread-safe exchanges. The approach meets the requirement of precisely replicating the Python's HID logic by ensuring the same sequence of reports is sent and received, just implemented in C++ with careful control over timing and memory.

4.2 Networking Module Design

Responsibilities: Manage all networking (UDP multicast/unicast), ensuring non-blocking operation and correct protocol implementation.

Implementation Class: `class NetworkModule`. Key members:

- **Socket Handles:** We will likely have two `SOCKET` members:
    - `multicastSock` for receiving broadcast/multicast data (if the protocol uses a group to send state).

    - `unicastSock` for sending and/or receiving directed commands (e.g., receiving direct responses or sending control messages to the simulator host).
      If the Python app used only one socket for both, we might do similarly, but many simulation setups use one port for inbound state (multicast or broadcast from sim) and another for outbound commands.
- **Initialization:** `bool InitNetwork()`. This performs:
    - `WSAStartup()` to initialize Winsock (at app start).
    - Create sockets with `WSASocket(AF_INET, SOCK_DGRAM, 0, NULL, 0, WSA_FLAG_OVERLAPPED)`.
    - Set socket options: `setsockopt(SO_REUSEADDR)` if needed for multicast, `setsockopt(IP_MULTICAST_LOOP)` possibly off to not receive our own sends, `setsockopt(IP_MULTICAST_TTL)` if needed, and `setsockopt(SO_RCVBUF)` to a decent size. Also, `bind()` the receive socket to the multicast port (and possibly address INADDR_ANY if we want to receive on any interface). Use `setsockopt(IP_ADD_MEMBERSHIP)` to join the multicast group as specified in config (with the local interface address or default).
    - If a separate unicast socket is used for sending, it might not need binding (we can just use sendto with a target). But if expecting replies on it, we bind it too.
    - Create an event object or use `WSACreateEvent()` for each socket's overlapped operations.
- **Thread Function:** `NetworkModule::NetworkThreadFunc()`. This will wait for incoming data and handle outgoing requests. There are two main patterns to consider:
    1. **Event-based Wait (multiple events):** We can post one overlapped receive on each socket and then wait on both events.
        - Use `WSARecvFrom(multicastSock, ...)` with `OVERLAPPED overlap1` that has `hEvent = event1`. Similarly `WSARecvFrom(unicastSock, ...)` with `overlap2/event2`.
        - Then, in a loop, use `WSAWaitForMultipleEvents(2, events, FALSE, INFINITE, FALSE)` to wait for any socket to signal data readylearn.microsoft.com.
        - Determine which event signaled (`WSAWaitForMultipleEvents` returns an index), then call `WSAGetOverlappedResult()` for that socket to get the bytes and flags. Process the received buffer accordingly.
        - After processing, immediately repost another `WSARecvFrom` on that socket (so we don't miss further packets).
        - Also handle a shutdown event similarly as in HID (include it in the wait array or check an atomic flag periodically).
    2. **IOCP-based:** Associate both sockets with a single IOCP. Then `GetQueuedCompletionStatus` will yield completions for either. In practice, for just two sockets, the event method is simpler and sufficient. We'll proceed with the event method for clarity.

Pseudo-code for event loop:

```
cpp
CopyEdit
```

```
WSAOVERLAPPED ovMulticast = {}, ovUnicast = {};
ovMulticast.hEvent = WSACreateEvent();
ovUnicast.hEvent   = WSACreateEvent();
// Post initial receives
WSARecvFrom(multicastSock, &wsaBuf1, 1, NULL, &flags1,
(sockaddr*)&srcAddr1, &srcLen1, &ovMulticast, NULL);
WSARecvFrom(unicastSock,   &wsaBuf2, 1, NULL, &flags2,
(sockaddr*)&srcAddr2, &srcLen2, &ovUnicast, NULL);

HANDLE waitEvents[3] = { ovMulticast.hEvent, ovUnicast.hEvent,
shutdownEvent };
while (!shutdownRequested) {
    DWORD idx = WSAWaitForMultipleEvents(3, waitEvents, FALSE,
INFINITE, FALSE);
    if (idx == WAIT_OBJECT_0) {
        // Multicast socket received data
        WSAGetOverlappedResult(multicastSock, &ovMulticast,
&bytesRecvd, FALSE, &flags1);
        ProcessReceivedPacket(recvBuffer1, bytesRecvd, srcAddr1);
        // repost receive
        ResetEvent(ovMulticast.hEvent);
        WSARecvFrom(multicastSock, ... &ovMulticast ...);
    }
    else if (idx == WAIT_OBJECT_0+1) {
        // Unicast socket data
        WSAGetOverlappedResult(unicastSock, &ovUnicast, &bytesRecvd,
FALSE, &flags2);
        ProcessReceivedPacket(recvBuffer2, bytesRecvd, srcAddr2);
        ResetEvent(ovUnicast.hEvent);
        WSARecvFrom(unicastSock, ... &ovUnicast ...);
    }
    else if (idx == WAIT_OBJECT_0+2) {
        // shutdownEvent signaled
        break;
    }
}
```

This demonstrates a non-blocking loop that handles both sockets and uses events so the thread sleeps when no data. We ensure to reset events and reuse them. The `ProcessReceivedPacket` will contain our application logic for incoming messages.

- **Incoming Packet Processing:** `void ProcessReceivedPacket(char* data, int length, sockaddr_in& source)`. This function parses the data according to our protocol:
    - If the data is a state update from the simulation (e.g., a series of values), it might need to update outputs on the HID. For example, in a DCS-BIOS-like protocol, the data might contain address-value pairs for cockpit indicators. The Network module could translate those into appropriate HID outputs or internal state. In such a case, it might call `HIDModule::SendOutputReport` for each relevant change (or batch them if needed). The logic will be built to mirror the Python code (for instance, if Python filtered or processed this data before acting).

    - If the data is a direct command (inbound unicast), it might instruct a specific action. E.g., "Reset device" or "Calibrate". The handler would then call into HID or other modules accordingly.
    - Robustness: The parser will validate packet length and content (since we avoid exceptions, we rely on explicit checks). If an unknown or malformed packet is received, we log it and ignore to maintain stability.
    - Since processing is done in the network thread, which is high priority, we should keep it efficient: likely just decoding a few bytes and calling another module. Any heavy computation or long loop should be avoided (though in our context, packets are small and processing trivial).
- **Outgoing Packets:** The module will provide methods or internally send data out:
    - For periodic broadcasts (if any) – e.g., maybe the device state needs to be sent at intervals – the Network thread can use a timer (like `WaitForSingleObject` on a timed event or just a non-blocking sleep loop) to trigger sends. Bounded periodic tasks can be integrated by checking time stamps within the loop. But if the Python app only sent on events, we will do the same to avoid unnecessary traffic.
    - On an event from HID (like a switch change), if we decided on direct cross-calling, the HID thread calls `SendPacket` in network. If instead we had queued it, the network thread would pick it up. Assuming direct call, our `SendPacket` function would do:
        - Construct a `sockaddr_in` for destination (from config, e.g., simulator IP and port).
        - Possibly perform some protocol encoding (pack the message bytes).
        - Call `sendto(socket, buf, len, 0, (sockaddr*)&dest, sizeof(dest))`. Check for errors: in normal case, it returns immediately. If it returns SOCKET_ERROR, get `WSAGetLastError`. If it's WSAEWOULDBLOCK (should not usually happen with UDP), we could log and drop or try later. Most other errors we log (e.g., network unreachable).
        - No blocking occurs here. If an error suggests the socket is dead (WSAENETRESET etc.), we might reinitialize it.
        - This function will be made thread-safe if called from HID thread: e.g., use a mutex around any use of `dest` structure if it's shared or any sequence numbers. But likely not needed if just sending raw.
    - If using IOCP, sending can also be overlapped, but given the simplicity, we may not do overlapped send; the small chance of blocking in send (if kernel buffer is full) is negligible for small UDP and can be mitigated by setting a large send buffer. We will assume safe to use blocking sendto, which typically doesn't block for UDP unless extreme conditions.
- **Cleanup:** On shutdown, we set `shutdownEvent` so the wait loop breaks. Then:
    - Call `CancelIoEx` on each socket with the overlapped to cancel any pending recv, and/or close the sockets which will also cause WSAWait to return (with error we handle).
    - After thread exits, call `WSACleanup()`.
    - If any threads might still use network (shouldn't after flag), ensure they stop before cleanup.

- **Headers/Libraries:** We will include `<winsock2.h>` and `<ws2tcpip.h>` for address conversions if needed. Link with `Ws2_32.lib`. We ensure to call `WSAStartup` early (in core init) because Winsock functions require that. We also define the project to be multithreaded, which is default for Win32 C++.

Deterministic Networking: Using overlapped I/O and events as above gives a deterministic event-driven network handling. There's no polling interval to tune – it reacts as fast as possible to incoming packets. If multiple packets arrive, the completion events queue up; the moment we finish processing one, the next is handled (Winsock will signal the event again or multiple events can queue, though `WSAWaitForMultipleEvents` will only indicate one at a time; we loop quickly to handle possibly back-to-back signals). This ensures **bounded latency** from packet arrival to handling, typically on the order of the thread's scheduling latency (which at high priority is very low). By not using `recvfrom` in a blocking or polling loop with timeouts, we don't risk missing anything or wasting time sleeping.

Network Protocol Consideration: If reliability or ordering is an issue (UDP is not reliable), the Python app might have included checks or sequence numbers. We will maintain those if present. For example, if packets contain sequence IDs and the Python code filtered duplicates or out-of-order data, we will implement the same in `ProcessReceivedPacket`. The module can store last sequence numbers in static variables to detect repeats or drops (and perhaps log if something is missed for diagnostics).

In conclusion, the NetworkModule is engineered for **efficiency and responsiveness** using core Winsock2 capabilities. It will be thoroughly tested under expected load (for example, if the simulator sends 30 updates per second, we ensure our thread can handle that for hours without dropping packets or increasing latency).

4.3 GUI Module Design

Responsibilities: Present a minimalistic interface to the user and allow basic control of the application (e.g., start/stop or config view). Display status of HID and network and possibly logs.

Implementation Approach: We will implement the GUI using the Win32 API directly, as a standard Windows application (likely a **Windowed application with WinMain** entry point). We will avoid using heavy frameworks, but for convenience, we might use a dialog template for layout if the interface is very simple (e.g., one dialog with controls). However, since we want to compile statically and avoid MFC, it might be simplest to create a window class and manually place controls with `CreateWindow` for each child control.

Main Window: Could be a small `HWND` with title "Cockpit Controller". We register a window class (`WNDCLASS`) with our `WndProc` callback. In `WM_CREATE`, we create:

- Static text controls for labels like "Device Status:" and a dynamic text control next to it to show "Connected" or "Disconnected".
- Possibly an LED indicator (we could simulate with a colored static or an icon).
- Buttons like "Reconnect" (if manual reconnect is desired), or "Exit".

- A multiline `EDIT` control in read-only mode for logs (if we choose to show logs in the GUI). This control can be populated with text via `SetWindowText` or appended via `Edit_SetText` messages periodically.
  We will position these using `CreateWindowEx` with styles WS_CHILD|WS_VISIBLE.

Message Loop: In `WinMain`, after creating the window, we run `MSG msg; while(GetMessage(&msg, NULL, 0, 0)) { TranslateMessage(&msg); DispatchMessage(&msg); }`. This is a standard GUI loop that sleeps when no message (so GUI thread uses negligible CPU when idle).

Interaction with Core: The GUI needs to reflect internal state. We plan the following:

- The HID and Network modules (or core) can post messages to the GUI window to notify status changes. For example, if device gets disconnected, the HID thread (which cannot directly call GUI functions, since GUI calls must be in GUI thread) can do `PostMessage(hwndMain, WM_APP_DEVICE_CHANGED, newStatus, 0)`. We define a custom `WM_APP+1` message for "device status". In `WndProc`, we handle it by updating the status text control (e.g., set text to "Disconnected" in red).
- Similarly, a `WM_APP_NETWORK` message could indicate network errors or connect status.
- The GUI might have a timer (`SetTimer`) to periodically poll some stats. For instance, every second, query an atomic counter of packets or logs. Or we can simply rely on posted events as above to minimize overhead.

User Commands:

- If there's a "Reconnect" button, clicking it (`WM_COMMAND` with that button ID) should initiate device reconnection. The GUI would call a core function like `Core::RequestReconnect()`, which signals the HID thread to attempt re-opening the device if not connected. This call must be thread-safe; likely just set an atomic flag that HID thread checks, or wake HID thread via an event. Because the GUI and HID are separate threads, we avoid direct calls that require waiting. Instead we might do `SetEvent(hidReconnectEvent)` that the HID thread waits on (or incorporate into its loop).
- An "Exit" button or menu will post `WM_CLOSE`, which we handle by posting the shutdown event to threads and then destroying the window (leading to message loop exit).

No Exceptions, No Heap in GUI: The GUI code will mostly use local stack variables (for text buffers etc.). GDI calls like `TextOut` or static text don't allocate long-term. We should be mindful that `SendMessage` and such might allocate internally but that's OS-managed. We won't use C++ new in GUI either; not needed for such a simple UI. Also, we disable C++ exceptions globally, so any failure (like `CreateWindow` returning null) we handle by returning an error code or using an `if` to check and then gracefully exit (perhaps log and `MessageBox` an error to user).

Refresh and Painting: For static controls and edit controls, the OS handles painting. If we had custom drawing, we would handle `WM_PAINT`, but likely not needed except maybe to draw a colored circle indicator for device status. Even that could be an icon or using `Static` with

`SS_ICON`. We could load a green/red icon from resources for status. If doing custom, we ensure to use `BeginPaint/EndPaint` properly. In any case, these operations are not performance-critical (GUI is low priority and infrequent updates).

Threading Consideration: Only the main thread will access GUI elements (requirement of Win32). Other threads will only communicate via `PostMessage` or similar – which is thread-safe. We must ensure not to block the GUI thread waiting on other threads in a way that could cause deadlocks. The GUI should remain responsive. For example, if a user hits "Exit", we might set a flag and wait for threads to terminate before closing the app. But we should perform that wait outside the GUI message loop (maybe in WinMain after `PostQuitMessage`). We might initiate shutdown on WM_CLOSE by signaling threads, then in WinMain after the loop, join threads. That means when WM_CLOSE happens, we do:

```
cpp
CopyEdit
case WM_CLOSE:
    core.InitiateShutdown(); // sets flags and events for threads
    // maybe show a "shutting down" message or disable controls
    return 0; // (we don't destroy window immediately, or we do and still 
wait)
```

Actually, perhaps easier: call `DestroyWindow` and then outside the loop, after GetMessage returns false (which happens after WM_QUIT is posted), we call `join` on threads. This way, the window is closed (not responsive to user anymore) but we finalize threads properly before exiting process. This ensures no threads linger.

Visual Studio Project Settings for GUI: We will compile as a Windows GUI subsystem (`/SUBSYSTEM:WINDOWS`) since we have a WinMain (not console). No console will appear (which is desired for an end-user app). We'll statically link runtime (`/MT`) so the exe doesn't require MSVC runtime DLLs. We'll specify no RTTI `/GR-` and no exceptions as globally (so it affects GUI code as well).

In summary, the GUI module is straightforward but crucial for user confidence and control. It will be developed with attention to not interfering with the real-time threads. Its design replicates the Python GUI minimalism, likely improving performance (native code vs Python UI library) and giving a native Windows look.

4.4 Configuration Module Design

Responsibilities: Load `settings.ini` at startup and provide configuration parameters to other modules.

Implementation: Likely a simple singleton class `Config` or just a namespace with global variables populated on init. Since configuration is read-only after init (no need to modify at runtime except via editing the file and restarting app), a simple approach suffices.

- We will use **inih** library for parsing:

    - Include `ini.h` and `ini.c` in the project (compiled as part of our code). No separate lib needed.
    - Define a callback function `int IniHandler(void* user, const char* section, const char* name, const char* value)` that will be called for each entrygithub.com. We can map known section/name to our variables. For example:

      ```
      cpp
      CopyEdit
      struct Settings {
          char deviceVID[5];  // maybe as string "0x1234"
          unsigned deviceVidVal;
          unsigned devicePidVal;
          char multicastIP[16];
          int multicastPort;
          char unicastIP[16];
          int unicastPort;
          // ... other fields
      } settings;
      ```

      In IniHandler:

      ```
      cpp
      CopyEdit
      if (strcmp(section, "Device") == 0) {
          if (strcmp(name, "VID") == 0) settings.deviceVidVal =
      strtol(value, NULL, 16);
          else if (strcmp(name, "PID") == 0) settings.devicePidVal =
      strtol(value, NULL, 16);
      } else if (strcmp(section, "Network") == 0) {
          if (strcmp(name, "MulticastGroup") == 0)
      strncpy(settings.multicastIP, value,
      sizeof(settings.multicastIP));
          else if (strcmp(name, "MulticastPort") == 0)
      settings.multicastPort = atoi(value);
          ...
      }
      ```

      This is straightforward. We ensure to use safe conversion (strtol, etc.) and check ranges (e.g., port 0-65535).

    - Call `ini_parse("settings.ini", IniHandler, &settings)` at startup. If it returns < 0, file not found or parse error; we handle that by perhaps using defaults or erroring out.
    - We will compile inih with `INI_USE_STACK=1` (default) so it uses a fixed 200-byte line buffer on stackgithub.com. If any line exceeds that (unlikely in an INI), it will error – we can increase `INI_MAX_LINE` define if needed to, say, 512.
- Once parsed, the Settings struct is ready. We might then do some post-processing:
    - Convert the human-friendly values to internal ones (like store `sockaddr_in` for the addresses for faster use, prepare wide-char device interface GUID if needed, etc.).

    - For instance, we can call `InetPton(AF_INET, settings.multicastIP, &multicastAddr.sin_addr)` to get binary IP.
    - This can all be done once in core init.
- **Accessing Config:** Modules can either access the global `settings` struct directly (if kept in a common header) or via getters `Config::GetMulticastIP()` that returns the value. Given our context and need to avoid complexity, a global struct (declared `extern` in a header and filled in at init) might be simplest and acceptable (as it's const after init).
- **Thread Safety:** The config is only written at startup (before threads launch), and then only read. So no concurrency issues. To be safe, we could mark the struct `const` after init to prevent accidental modifications.
- **No dynamic memory:** inih itself doesn't allocate heap unless large lines or if we turned on heap mode. We keep it on stack mode. Our Settings struct is static. So all good.
- If no external library was allowed, we could do manual parsing:
    - Use `CreateFile("settings.ini")` and read the whole file into a static char buffer (we know its size will be bounded, say 4KB).
    - Then parse line by line using `strtok` or manual state machine (skip comments, find '=', etc.). This is doable but since inih is already a robust solution, we prefer it to avoid reintroducing parsing bugs.

Example settings.ini (for clarity in document, maybe include a snippet):

```
ini
CopyEdit
[Device]
VID=0x1234
PID=0xABCD

[Network]
MulticastGroup=239.255.0.1
MulticastPort=5000
UnicastIP=192.168.0.100
UnicastPort=6000
```

Our parser will handle both hex and decimal as shown. It's also case-insensitive by default for key names (inih can be configured but usually it treats names case-sensitively by default). We'll document that in config if needed (or just handle either case in code).

By maintaining the INI format, we ensure continuity for users upgrading from the Python version – they can use their existing config file without changes.

4.5 Logging and Statistics Module Design

Responsibilities: Provide a lightweight logging mechanism for debugging/monitoring, and gather statistics (counts, timings) without impacting performance or memory.

Design: We can implement this as a collection of **circular buffers** protected by atomic indices (lock-free) for each thread or each log level. However, a simpler approach given moderate log volume is a single circular buffer with a small critical section.

- **Data Structure:**

  ```
  cpp
  CopyEdit
  struct LogEntry {
      uint32_t timestamp_ms;  // or high-res timestamp truncated
      LogLevel level;
      ModuleID source;
      char message[64];  // fixed-size message text
  };
  static LogEntry logBuffer[LOG_CAPACITY];
  static std::atomic<uint32_t> logWriteIndex;
  static uint32_t logReadIndex; // for reading (only used by GUI thread
  likely)
  static const uint32_t LOG_CAPACITY = 1000;
  ```

  We choose message length 64 (adjust as needed for typical log). LOG_CAPACITY maybe 1000 entries (fits in a few hundred KB at most). This is all static.

- **Logging API:**

  ```
  cpp
  CopyEdit
  void Log(LogLevel lvl, ModuleID src, const char* fmt, ...);
  ```

  This function would use a `static char temp[64]` to format the message via `vsnprintf`. We will avoid heap by using this static or thread-local buffer. Then we obtain an index:

  ```
  cpp
  CopyEdit
  uint32_t idx = logWriteIndex.fetch_add(1, std::memory_order_relaxed);
  idx %= LOG_CAPACITY;
  // Overwriting oldest if overflow (logReadIndex could be maintained if
  we want to know new vs old)
  LogEntry& entry = logBuffer[idx];
  entry.timestamp_ms = GetTickCount() & 0xFFFFFFFF; // or a similar timer
  entry.level = lvl;
  entry.source = src;
  strncpy(entry.message, temp, sizeof(entry.message));
  entry.message[sizeof(entry.message)-1] = '\0';
  ```

  This simple scheme may cause race conditions if two threads write the same entry simultaneously (since they may compute idx the same before increment? But fetch_add is atomic and returns unique values, so they won't collide on idx assignment; however, if logWriteIndex wraps after LOG_CAPACITY, entries might override before read – but that's expected in circular buffer). We accept that some logs might get overwritten if writing is faster than reading; that's typical behavior.

  - We could add a sequence number or increment a generation in each entry to help GUI know if it missed entries, but not strictly needed.

    - The GUI can maintain a last seen index to only process new logs.
- We ensure `Log` is async-signal-safe style (no dynamic mem, no waiting). It just does an atomic increment and some copying. That should be fine to call from any thread, even high-priority, since it does not block (worst-case, if two logs happen at same moment, they both do fetch_add (which uses atomic CAS internally) but that's short, likely a few CPU cycles, which is fine).
- **ModuleIDs and Levels:** Define an enum for modules (e.g., 0=CORE,1=HID,2=NET,3=GUI) and levels (INFO, WARN, ERROR, DEBUG). This helps filter if needed. We can decide to not log INFO from high-rate events to avoid flooding.
- **Usage:** Throughout the code, instead of `printf` (which we won't use in a GUI app anyway), we call `Log()`. For example:

  ```
  cpp
  CopyEdit
  if (bytesRead == 0) {
      Log(WARN, HID, "Device returned 0 bytes (possible disconnect)");
  }
  ```

  Or

  ```
  cpp
  CopyEdit
  Log(INFO, NET, "Received %d bytes from %s:%d", bytesRead,
  inet_ntoa(srcAddr.sin_addr), ntohs(srcAddr.sin_port));
  ```

  This helps in debugging. In release builds, we might disable verbose logs via macros if needed to optimize, but the overhead is quite low anyway and bounded.

- **Statistics:** We will maintain counters like:
    - `std::atomic<uint64_t> packetsSent, packetsReceived;`
    - `std::atomic<uint32_t> hidReportsSent, hidReportsReceived;`
    - `std::atomic<uint32_t> errorCount[MODULE_COUNT];` etc.
      These are incremented in code where relevant. For instance, in `ProcessReceivedPacket`, do `packetsReceived++`. In HID `HandleInputReport`, do `hidReportsReceived++`. These counters can be displayed in GUI or printed in logs periodically.
      We choose atomic to allow increment from multiple threads (e.g., both network and HID might increment errorCount concurrently). Alternatively, each module can keep its own counter in its thread and maybe the core reads them with appropriate sync (since reading atomic is fine, we can just use atomic for simplicity).
      They are static or global but that's okay given their small size.
- **Memory Impact:** The log buffer as defined is fixed 1000 * (approx 72 bytes) ~ 72kB, trivial for a PC. Stats counters are negligible. So memory is fixed and small.
- **Thread safety & Reading:** The GUI (or a testing thread) can read logs by checking where the write index is. If GUI wants to pop all new logs:

```
cpp
CopyEdit
static uint32_t lastReadIdx = 0;
while (lastReadIdx < logWriteIndex) {
    LogEntry& e = logBuffer[lastReadIdx % LOG_CAPACITY];
    DisplayLogEntry(e);
    lastReadIdx++;
}
```

But since logWriteIndex is atomic and can wrap, we mod it accordingly. We might use a simpler approach: GUI each time reads all entries from `lastGUIShown` to `logWriteIndex-1` (mod capacity), taking into account wrap if `logWriteIndex` has overtaken by more than capacity (meaning it lapped and we missed some, which we could detect if `logWriteIndex - lastGUIShown > LOG_CAPACITY`, then we note "some logs dropped").
This complexity is acceptable given log is mainly for human debugging and not mission-critical.

- **Alternative:** For absolute simplicity, we could avoid even atomic ops by having separate log buffers per thread and the GUI merges them. But merging is complex and multi-thread reading still needed. The single buffer with atomic index as above is a known pattern for lock-free logging with minor trade-offsembeddedartistry.com.
- We will also implement a mechanism to log to file if needed (for long runs). Perhaps on exit, dump the ring buffer to a file on disk. But since requirement doesn't mention file logging, we may omit or leave as an optional.

This Logging module ensures that we **never allocate memory for logs**, and writing a log is **O(1) constant time**. This aligns with performance goals. It's also decoupled: if GUI is slow or not open, logs still get recorded (overwriting old as needed) without blocking the source threads. This decoupling is crucial for fault containment – e.g., if GUI hangs or is busy, the real-time threads are not affected.

5. Code Patterns and Best Practices in Implementation

In building this C++ application, we will adhere to patterns that enhance safety, clarity, and performance. Below are some key practices and examples:

5.1 Language and Library Usage

- **Modern C++ (C++20) features where appropriate:** We will use `constexpr` for compile-time constants and calculations, `enum class` for type-safe enumerations (e.g., for ModuleID or LogLevel), `std::array` for fixed-size arrays (for instance, an array of 8 bytes for an HID report, rather than raw C array, to get bounds-checking in debug builds). However, we avoid features that rely on runtime allocation or extensive template instantiation that may bloat code. We disable RTTI, so no use of `typeid` or dynamic_cast; our design doesn't require them. We disable exceptions, so we'll ensure to not use any standard library component that might throw exceptions during normal operation (for example, avoid using `std::vector::at` which throws on out-of-range; use `operator[]` with our own checks).

- **No use of new/delete at runtime:** If any dynamic allocation is needed, we'll do it at startup and never free (so no fragmentation). For instance, if using `std::thread`, it allocates internally for thread local storage; we just accept that at startup. If using any third-party (like HIDAPI), we'll check that it doesn't spawn threads or allocate frequently; likely it just allocates some memory when opening a device. We will document any such usage and ensure it's not in a loop.
- **Standard Library vs. Win32 API:** We choose the lower-level API when it gives more control. For example, for threading, `std::thread` is convenient but we might prefer `CreateThread`/`SetThreadPriority` to immediately set priorities and get a handle to use with `WaitForSingleObject` on shutdown. We can still use `std::thread` if we then use OS calls to adjust the thread's priority (we can get the handle via native_handle from std::thread and call SetThreadPriority). However, since exceptions are off, we must be careful: `std::thread`'s constructor throws on failure to create thread. We can wrap that in a try/catch(...)/std::terminate, or better, use `_beginthreadex` or `CreateThread` which returns an error code instead. We will likely use Win32 directly for threads to avoid any C++ runtime issues.
- **Deterministic destruction**: Although exceptions are off, we still rely on destructors for cleanup of objects when they go out of scope normally. For example, we may use RAII wrappers for critical handles:
  - A small class `AutoHandle { HANDLE h; ~AutoHandle(){ if(h) CloseHandle(h);} }` to ensure events or thread handles close automatically in all paths. Since no exceptions, the only thing to consider is normal scope exit vs. program termination, but it's good practice.
  - Similarly, a RAII class for WSAStartup: e.g., `struct WSAInit { WSAInit(){ WSAStartup(...);} ~WSAInit(){ WSACleanup();} };` used in main. This reduces risk of missing cleanup.
- **Header organization:**
  - Use `<windows.h>` (with lean and mean) in CPP files where needed, not in headers, to avoid polluting namespace globally.
  - Create headers for each module (`HIDModule.h`, `NetworkModule.h`, etc.) that expose needed interfaces and hide internal details.
  - We will forward declare where possible to speed compile times and reduce dependencies (especially since including windows.h everywhere can slow compilation).
  - Put compile-time constants and config (like buffer sizes) in a header or `constexpr` in appropriate scope so that changes are centralized.
- **Compiler Settings:** We will set compiler flags for optimization and safety:
  - `/O2` or `/Ox` for a high level of optimization in release builds.
  - `/Oi` to enable intrinsic functions (e.g., using intrinsics for some calls which might improve speed).
  - `/Og` (global optimizations) and `/GL` (link-time code generation) could be used to allow cross-module inlining and optimize the whole program – beneficial because we have many small functions that could be inlined across translation units. We must pair /GL with `/LTCG` in linker.

- `/Qspectre-` if not needed (just to not include mitigations overhead if not needed, though this is minor).
- Use `/Zo` or `/DEBUG` in debug builds to get good debug info, and disable optimizations in debug to ease step-through (but of course debug build won't meet realtime constraints, that's fine).
- `#pragma comment(lib, "Ws2_32.lib")` and others can be included so linking is automatic.
- Set the `/MT` flag to link the static runtime (for both release and debug we'll use /MT and /MTd respectively)stackoverflow.com.
- `/NXCompat`, `/DynamicBase` are on by default for security (ASLR, DEP).
- Disable `/GS`? In most cases, we can keep buffer security checks (/GS) on since it adds minimal overhead but catches stack overruns. For aerospace, sometimes they turn it off to avoid any runtime checks, but buffer overruns are not expected if code is correct. We prefer to keep /GS on to be safe, it only affects unsafe functions usage which we will minimize anyway.
- `/GR-` to disable RTTI as said, and in project settings disable C++ exceptions (which removes /EHsc).
- Possibly use `/kernel` flag as an alternative means to disable exceptions/RTTI as noted by some sourcesstackoverflow.com, but that's intended for driver-like behavior. We might avoid it because it also defines `_KERNEL_MODE` which could have other effects. We'll stick to explicit flags.

5.2 Error Handling and Fault Containment Patterns

Without exceptions, our error handling will be explicit and localized:

- Every call that can fail (Win32 APIs, socket calls) returns an error code that we must check. The code will follow a **C-style** pattern:

```
cpp
CopyEdit
if (!DeviceIoControl(...)) {
    DWORD err = GetLastError();
    Log(ERROR, HID, "DeviceIoControl failed, err=%lu", err);
    // handle or propagate error
    return ERR_DEVICE_FAILURE;
}
```

  We will propagate errors up only when necessary. For instance, if HID `OpenDevice()` fails, we might return an error code to core, which then could decide to retry or show GUI message. Within threads, often the error is handled by retry loops or graceful exit of that thread.

- We will define some error codes or use HRESULT/Win32 codes directly. Possibly an `enum Status { OK=0, ERR_TIMEOUT, ERR_DISCONNECTED, ... };` for internal use.
- **Local Recovery:** If a network send fails transiently (say due to buffer), we can retry after a brief pause or simply drop (UDP doesn't guarantee delivery anyway). If a receive fails

due to network disconnection, we may attempt to reopen sockets after a delay. The network thread can sleep a few seconds and try `InitNetwork` again, logging attempts, until success or shutdown. This way a temporary outage doesn't bring down the process.

- **Watchdog / Monitoring:** In aerospace software, sometimes a watchdog monitors threads. We could implement a simple heartbeat: each thread updates a timestamp in a shared struct periodically (say every loop iteration or every second). The core thread could check if any thread has not updated in a long time (indicating it hung) and log a critical error or attempt restart. Implementing a full restart of a hung thread is complex (we'd rather avoid the hang in first place by design), but detection can help in testing.
- **No use of `assert` in production** (since that calls abort). Instead, for conditions that "should not happen," we will log an error with details. If it's something unrecoverable (like memory corruption detected or an impossible state machine condition), we might still decide to shut down gracefully rather than continue with bad state. In that case, the core could receive a signal and initiate shutdown. For example, if HID thread finds the device keeps giving malformed data (indicative of a serious bug), after X occurrences it could log and signal core to stop the program. In an aerospace context, fail-fast might be preferred over uncontrolled behavior.
- **Testing and Verification:** We will incorporate test code (in debug builds) to simulate faults, such as forcing a device disconnect event or network loss, ensuring our error handling paths work correctly. These won't be in the final compiled but used during dev.

5.3 Deterministic Scheduling and Real-Time Considerations

To further guarantee bounded timing:

- **Thread Priorities:** As mentioned, set critical threads to high or realtime. Code example:

  ```
  cpp
  CopyEdit
  HANDLE hHIDThread = CreateThread(NULL, 0, HIDThreadFunc, 0, 0, NULL);
  SetThreadPriority(hHIDThread, THREAD_PRIORITY_TIME_CRITICAL); //
  highest real-time level in user mode
  ```

  Also:

  ```
  cpp
  CopyEdit
  SetPriorityClass(GetCurrentProcess(), HIGH_PRIORITY_CLASS);
  ```

  If we find it necessary, `REALTIME_PRIORITY_CLASS` for process and `SetThreadPriority(hHIDThread, 15 or above)` can be done, but this requires administrator rights typicallyscorpiosoftware.netlearn.microsoft.com. We will document that if truly needed. Likely HIGH priority (which corresponds to base priority 13) is enough, as it's above normal OS services but below realtime.

- **Affinity:** If the target machines are multi-core, we might dedicate one core to the HID and Network threads. This can eliminate contention with other OS activities. E.g.,

```
cpp
CopyEdit
SetThreadAffinityMask(hHIDThread, 1 << 2); // run HID thread on CPU
core 2
SetThreadAffinityMask(hNetworkThread, 1 << 2); // maybe same core 2 if
not heavy, or use 3
```

However, Windows already load-balances, and if our threads are high priority they'll preempt others anyway. Affinity might help consistency if we worry about other interrupts on that core. It's an optional tweak.

- **Timer usage:** We avoid using `Sleep()` for any timing because Sleep has a resolution that depends on the system timer and could be 1-15ms (unless high-resolution timer is enabled). Instead, where we need periodic behavior, we either use Windows timers (which are message-based and not super precise either) or a busy wait on a high-res counter in a low priority thread (not good for CPU though). For high precision, Windows offers `QueryPerformanceCounter` for timestamping but not for sleeping. Since our design is mostly event-driven, we minimize timed waits. For example, if we need to ensure the HID thread doesn't poll too fast, we could add `Sleep(1)` inside a reconnect retry loop to yield CPU, but at high priority, Sleep might not be accurate. We can use `WaitForSingleObject` with timeout for that (which is similar). It's fine because during reconnect attempts we're not time-critical (device is absent anyway).
- **I/O Buffering:** We will set sufficient buffer sizes to avoid blockage. E.g., setsockopt SO_SNDBUF and SO_RCVBUF to some tens of KB to ensure the kernel can buffer a few packets if our thread is momentarily busy. For HID, not much we can do on buffering, just ensure we read promptly.
- **Testing bounded latency:** We will instrument with QueryPerformanceCounter to measure how long certain operations take (e.g., the gap between an input event and finishing its processing). These can be logged in debug builds to ensure our design meets the targets (likely microseconds to low milliseconds).

5.4 Class and Module Interfaces Summary

To tie together, here is a brief summary of the classes with important methods (pseudo-code style):

```
cpp
CopyEdit
// HIDModule.h
class HIDModule {
public:
    bool initialize();    // Open device, start thread
    void shutdown();      // Signal thread to stop and close device
    bool sendFeatureReport(const uint8_t* data, size_t len);
    bool sendOutputReport(const uint8_t* data, size_t len);
    // These post requests; actual sending done in HID thread context.

    // Status accessors:
    bool isDeviceConnected() const { return deviceConnected; }
    // ... maybe other stats like getLastError()
private:
    void HIDThreadLoop(); // The thread function
    void handleInput(const uint8_t* data, size_t len);
    // Device handles and internal state:
    HANDLE deviceHandle;
    OVERLAPPED ovRead;
    HANDLE hReadEvent;
    HANDLE hThread;
    bool deviceConnected;
    // Queues for outgoing reports:
    LockFreeQueue<Report> outQueue;
    HANDLE hOutEvent;
    // ...
};

// NetworkModule.h
class NetworkModule {
public:
    bool initialize();
    void shutdown();
    bool sendPacket(const uint8_t* data, size_t len, bool toMulticast);
    // possibly separate for different destinations

private:
    void NetworkThreadLoop();
    void processPacket(const char* buf, int len, const sockaddr_in& src);
    SOCKET sockMulticast, sockUnicast;
    WSAOVERLAPPED ovMulti, ovUni;
    HANDLE hEventMulti, hEventUni;
    HANDLE hThread;
    // dest addresses:
    sockaddr_in destSim; // for unicast sending
    // ...
};

// Logging (could be static functions in a namespace Logger)
namespace Logger {
    void log(LogLevel lvl, ModuleID src, const char* fmt, ...);
    bool getNextEntry(LogEntry& out); // for GUI to retrieve entries
    // internals: buffer, indices, etc.
}

// Core or Application
class CockpitControllerApp {
public:
    bool init();    // initializes Config, modules, GUI
    void run();     // enters GUI message loop
    void shutdown(); // called to begin graceful shutdown

private:
    HIDModule   hid;
    NetworkModule net;
    // possibly references to GUI window or handle
};
```

The **class diagram** conceptually: `CockpitControllerApp` contains a `HIDModule` and `NetworkModule` (composition). The GUI is not an object but rather the WinMain and WndProc functions, though we could wrap the GUI in a class as well if desired (e.g., `MainWindow` class that registers class and manages message dispatch). The relationships:

- App (core) calls `hid.initialize()` and `net.initialize()` during startup.
- HID and Network modules are largely independent, but the Network module likely needs a pointer/reference to HID module or a callback interface to invoke when it needs to send to device. We can handle that by passing a reference of HID to Network on init or by using a global function pointer. To keep it object-oriented, perhaps the NetworkModule has a pointer `hidModule` that it can call for sending outputs. Set this in App after constructing both: `net.setHIDModule(&hid)`. This avoids global and still decouples a bit. Alternatively, use a message in Logger or directly call static HIDModule::sendOutput (if those were static accessing a singleton). We will likely use a reference.
- Modules ensure not to call each other on the wrong thread in unsafe ways. The handshake between them is via safe functions or queues as described.

6. Build and Deployment Configuration

In this section, we list the specific project configuration and compiler/linker settings to achieve the requirements, as well as notes on how to build the application in Visual Studio for release deployment.

6.1 Compiler Settings (Visual Studio)

- **Platform Toolset:** Use the latest Visual Studio 2022 (v143) toolset or equivalent, target Windows 10 SDK.
- **Configuration:** We will have `Debug` and `Release` x64 configurations. In Release, we enable all optimizations.
- **C++ Language Standard:** Set to C++20 (/std:c++20) to allow modern language features that can improve clarity and possibly compile-time processing. However, we won't heavily rely on library features that aren't needed.
- **Code Generation Options:**
    - Runtime Library: **Multi-threaded** (/MT for Release, /MTd for Debug) to link statically to CRTstackoverflow.com. This avoids needing MSVC runtime DLLs and ensures more control (and slightly better performance since no DLL thunks).
    - Enable Intrinsic Functions: Yes (/Oi) to use intrinsic versions of some standard library calls (like `memcpy`) for speed.
    - Favor Speed over Size: Yes (e.g., /Ot) since performance is priority and our code size is small anyway.
    - Inline Function Expansion: Any suitable (/Ob2) to aggressively inline where beneficial (the compiler will decide).
    - Whole Program Optimization: Enable LTCG (/GL and linker LTO) in Release to allow cross-module inlining and optimization.

    - Floating Point Model: Since not much floating point expected, default is fine; if we had FP heavy code, we might use `/fp:fast` for speed, but careful if any precision issues. In this app, likely not significant.
    - Disable Incremental Linking (when using LTCG, it's off anyway).
    - **Security Checks:** Keep Buffer Security Check (/GS) enabled for safety (it adds canaries for stack buffers) – the performance impact is negligible and it adds protection against stack overflow exploitation.
    - Enable Function-Level Linking (/Gy) – typically default in release, to allow removal of unused functions.
- **Warnings and Static Analysis:**
    - Warning Level: /W4 (Level 4) or even /Wall. We will ensure code is warning-clean at /W4 at least. Some /Wall warnings can be excessive (especially from Windows headers), but we can selectively disable those. We treat warnings as errors (/WX) to enforce addressing them.
    - Static Analysis: Use Visual Studio Code Analysis (/analyze) periodically (not necessarily every build). Also consider running CppCoreCheck or clang-tidy with CERT or MISRA rules to catch any rule violations relevant to safety (this can be outside of build or as a separate configuration).
    - Disable Unneeded Features: Disable RTTI (/GR-)stackoverflow.com, and set "Enable C++ Exceptions" to No (which removes /EHsc and uses no exception handling). This combination ensures no C++ exception runtime support is linked.
- **Preprocessor Defines:**
    - `UNICODE` and `_UNICODE` (use wide-char WinAPI, which is standard on modern Windows).
    - `WIN32_LEAN_AND_MEAN` before including Windows headers to speed compile and reduce couplinglearn.microsoft.com.
    - Perhaps define `INI_USE_STACK=1` and `INI_MAX_LINE=256` (or appropriate) in project settings so that inih uses stack buffer and sets line length if needed.
    - We might define our own symbol like `AEROSPACE_MODE` to conditionally compile any extra safety features (for example, if we want to exclude some dev-only debugging code in production).
- **Linker Settings:**
    - Set SubSystem: Windows (/SUBSYSTEM:WINDOWS) for GUI app (so no console window).
    - Enable Link-Time Code Generation (/LTCG) to complement /GL.
    - Set large address aware (usually default on 64-bit).
    - Additional Dependencies: `Ws2_32.lib`, `Hid.lib`, `Setupapi.lib` (for SetupDi calls), `Cfgmgr32.lib` if using ConfigManager for device enumeration, and `User32.lib`, `Gdi32.lib` for GUI, which are usually linked by default in a GUI project. We add them if not.
    - Set a manifest for UAC if needed (we likely don't need admin rights unless using real-time priority – setting realtime priority might require admin or increased privilegescorpiosoftware.net; if so, user may have to run as admin or we include a manifest requesting `requireAdministrator` if absolutely needed for scheduling).
- **Optimization vs. Determinism:** We will test with optimizations on to ensure nothing breaks real-time assumptions (sometimes high optimization can re-order instructions in

ways that could affect timing, but since we rely on OS sync for actual ordering, it should be fine). We avoid undefined behavior in code, so optimization won't introduce unpredictability.

6.2 Recommended Header Inclusions

We outline the key headers for each part of the code:

- In **HIDModule.cpp**:

  ```
  cpp
  CopyEdit
  #include <windows.h>
  #include <hidsdi.h>    // HID driver API (requires including
  <windows.h> first)
  #include <setupapi.h>  // for device enumeration
  #include <cfgmgr32.h>  // (if needed for CM_Get_Device_Interface_List,
  optional)
  #include "HIDModule.h"
  #include "Logger.h"
  #include "Config.h"
  ```

  Also, we must link Hid.lib and Setupapi.lib accordingly.

- In **NetworkModule.cpp**:

  ```
  cpp
  CopyEdit
  #include <winsock2.h>
  #include <ws2tcpip.h>   // for sockaddr_in helpers and IP multicast
  options
  #pragma comment(lib, "Ws2_32.lib")
  #include "NetworkModule.h"
  #include "Logger.h"
  #include "Config.h"
  #include <windows.h>    // needed for WSAWaitForMultipleEvents (it's in
  Winsock2 too, but for some types)
  ```

  Note: winsock2.h should be included before windows.h or with lean and mean to avoid conflicts, but including windows first with lean and mean will also include winsock2 if WIN32_LEAN_AND_MEAN is defined. We ensure the include order doesn't break definitions.

- In **MainApp.cpp** (core or WinMain file):

  ```
  cpp
  CopyEdit
  #include <windows.h>
  #include "CockpitControllerApp.h"
  #include "HIDModule.h"
  #include "NetworkModule.h"
  ```

```
#include "Logger.h"
#include "Config.h"
#include <thread>  // if using std::thread for any launching (though
likely using CreateThread)
```

Possibly include <commctrl.h> or others if using common controls in GUI (not likely needed for just static and buttons).

- In **Logger.h**: just stdint.h or cstdint for fixed types, and atomic for logWriteIndex.
- In **Config.cpp**:

```
cpp
CopyEdit
#include <fstream> // if using file streams (but we might use C file or
CreateFile)
#include <cstring> // for strtok or manual parse
#include "Config.h"
#include "ini.h"   // if we use inih
```

plus any needed for conversion (like <stdlib.h> for strtol).

We maintain consistent include guards and minimal includes in headers to reduce coupling. For example, HIDModule.h can forward declare NetworkModule to avoid including its header (just declare class NetworkModule; if HID needs to call network, it can hold a pointer of type NetworkModule without including the full definition).

6.3 Deployment and Optimization Flags

For deployment (aerospace setting might require a very stable release build):

- We will build the **Release x64** configuration and test it thoroughly. That binary will be the one delivered.
- The executable can be tested under stress (lots of HID events, lots of network messages) to ensure CPU usage stays low (we expect perhaps <1% on a modern CPU when idle, and only small spikes on activity).
- Use **Performance Counters**: We might use Windows Performance Counter or ETW (Event Tracing) in test builds to measure event latency. Not included in final code, but as part of our verification step.
- Ensure **Deterministic Build**: We can use `/Brepro` (bitwise reproducible build) if required for certification (ensures no timestamp or address differences in binary across builds).
- **Optimization for size vs speed**: We lean speed. But if memory is not abundant, we note our binary will likely be a few hundred KB to a couple MB, which is fine on PC. We do static link CRT which adds some size (~100KB). It's acceptable.
- **Strip debugging symbols**: For release, we don't include PDB or we can generate one but not ship it if not needed. We might want a map file for reference if debugging issues at customer site with addresses.

- If multi-core performance is important, we consider enabling **SMT awareness**: e.g., avoid pinning two busy threads on the same physical core (this is more OS domain, but by default Windows will schedule on separate cores anyway if available).
- No use of **std::chrono** for sleeps (since we avoid sleeps), but if we needed a timeout, we'd use `std::this_thread::sleep_for` which is just a wrapper on Sleep. We avoid that as well in critical threads.

Finally, we ensure the application runs at **real-time priority** only if absolutely necessary and document that requiring admin privileges (because on Windows, setting real-time priority class might need it). If not, we use just below that to avoid requiring special rights.

7. Conclusion

This migration plan provides a structured blueprint to implement the CockpitController HID handler in C++ with an emphasis on performance, determinism, and reliability. By dividing functionality into focused modules (HID handling, network communication, GUI, etc.) and enforcing strict memory and thread management policies, the design meets aerospace-grade standards for software: it avoids dynamic memory and exceptions (preventing unpredictable delaysmedium.com), uses fixed-size buffers and circular logs for consistent memory usageembeddedartistry.com, and leverages Windows API features (overlapped I/O, high thread priorities) to achieve near real-time behavior. Each module is robust against faults in its domain, thereby containing errors without cascading failures.

The resulting C++ application will replicate all behaviors of the original Python program (HID report logic and network protocol) but with significantly improved performance and stability. Idle CPU usage will be effectively zero as threads wait on efficient synchronization objects rather than polling. Latencies for input/output processing are bounded and minimized through dedicated high-priority threads and direct asynchronous I/O. The system will shut down cleanly with proper synchronization, and resource cleanup and logging of any issues for traceability. All these measures align with the rigor expected in an aerospace or other safety-critical environment.

By following this design and the code patterns outlined, the development team can implement the new C++ CockpitController application confident that it will satisfy the strict requirements and operate reliably in real-world conditions. Each section of this document can serve as a reference during implementation and testing, ensuring that no aspect of performance or safety is overlooked. The end product will be a lean, efficient, and maintainable C++ program ready for use in demanding cockpit simulation or control scenarios.