white-label

middleware

App_EngineSteer v1.02

Technical Integrator's Manual

combined with Practical Guide

For internal engineering and system integration use.
Not for end-users.

Your brand — powered by fault-tolerant, time-saving white-label truth-sayers …

A truth-sayer is the software layer that decides what and when sensor data can be trusted.

Version 1.02 B010 • Generated 2026-05-31

Content

Content 2

1. Quick Start Checklist 5

2. Private Demo Videos on Request (*Broad Statement*) 6

3. How to Read this Document 7

4. Big Picture 8

(a) No Engine Control, No USV 8

(b) Too Many Engine Styles, One Integrator Brain 9

(c) *App_Enginesteer* as your Single Abstraction Layer 9

(d) Three Built-In *Simulation* Scenarios 10

(e) Private Demo Videos on Request 12

(f) *App_Enginesteer* as the "Missing Piece" 13

(g) Most Integration Teams Under-Estimate Engine Control at the Start 14

(h) One Control Box, "Many Engine Styles" 15

(i) Saving Your Team from Re-Inventing this Every Project 16

5. Engine Types 17

(a) Pure Mechanical Cable (Legacy but still Everywhere) 17

(b) Electro-Mechanical "Follow-Up" of Existing Head 18

(c) Analogue Throttle plus Separate Analogue Gear 18

(d) CAN-Bus / J1939 with RPM-based Control 19

(e) High-Level "Joystick / Vector" Systems 20

(f) Legacy Engine-Room Telegraph (For Completeness) 21

6. Designing the App 22

(a) Dedicated Harness 22

(b) Before the Wires are Connected 22

(c) Running the Demo 23

(d) Messaging Concept: 0xSYS / 0xTHR / 0xSTR 25

(e) Neutral Referee 26

(f) White-Label 26

(g) Safety & Integration Path 27

7. Overview 28

(a) Core Responsibilities 28
(b) Reliability by Design 29
(c) SIM vs REAL Operation 29
(d) Why This Matters 30
(e) Intended Use 30
8. Price 31
9. System Architecture 32
(a) High-Level Components 32
(b) Unpacking 33
(c) Installation & Directories 34
10. Licensing & Trial Behaviour 35
11. Updates 36
12. *config.txt* 37
13. *Log_App_EngineSteer_xxx.txt* 38
14. *..._heartbeat.txt* 38
15. *kill_App_EngineSteer.txt* 38
16. *restart_App_EngineSteer.txt* 39
17. *freeze_App_EngineSteer.txt* 39
18. Command Port Protocol (*not yet exposed in the demo app*) 40
(a) Core *GET / CMD / SET* commands 40
19. Squawk Port (not yet exposed in the demo app) 41
20. Test Buttons 42
21. Throttle and Steering Sliders 43
22. True Status Reporting / Commanding 44
23. SET / True Status / Cmd Reporting / Commanding 45
24. Reliability & Watchdogs 47
25. Known Limitations 47
26. Support 48
(a) Upgrade Path 48
(b) Branding Favicon 48
(c) Branding Splash Logo 49
(d) Software Updates 49

(e) Support Email 50
27. File Structure 51
28. Compatibility 52
29. Intended Audience 52
30. Errors & Omissions Excepted (E&OE) 52
31. FAQs 53
(a) Need for Simulation? 53
(b) Risk & Failure Handling? 53
(c) Integration & Time-to-Value? 54
(d) Control, Ownership & Lock-In? 55
(e) Scope, Boundaries & Responsibilities? 56
(f) What is the best way to use a *white-label* like yours? 57
(g) How does using your *white-labels* save time? 58
(h) What happens if the *white-label* fails, hangs, or stops sending data? 62
(i) How do I switch between SIM and REAL modes, and are the outputs identical? 62
(j) How do I see what the module is really doing internally? 63
(k) Can the *white-label* handle many channels / high update rates? 63
(l) How do the *white-label*s connect to my system, and how is this kept secure? 63
(m) How do we handle updates without breaking an existing vessel or project? 64
(n) Can we extend the module or add our own fields/logic? 64
(o) What exactly are we allowed to do with the *white-label* module? 65
(p) How do multiple *white-label* modules work together without conflict? 65
(q) How can we test and demo the system without an actual boat or sea trial? 66
(r) Safe Use and Distribution (Multiple) 67
32. Document Change History 68

1. Quick Start Checklist

- In the desired <drive>, at its root directory, create a directory called <***Apps***> (if you don't already have it).

- Copy the zipped file [sample name: ***App_xxx_Group.zip***] into this directory <***Apps***> and do an unpack.

- Locate and run ***App_EngineSteer.exe*** (as *administrator*) → [***config.txt***] is auto-created.

- At this point you can already run all 3 SIM scenarios without any **Engine** hardware.

- If you are asked to allow network access, please allow both private and public networks. You can then shut some of these access down when you are more certain about its access rights.

2. Private Demo Videos on Request (*Broad Statement*)

If you're evaluating *`App_EngineSteer`*, email us and we'll send links to show:

- Some engine videos created by the app
- Sample simulations
- Health of the logged device
- Fault-tolerance

3. How to Read this Document

This manual is intentionally dense. It is not meant to be read end-to-end in one sitting. It is intentionally layered so that the integrator can go deeper as and when needed.

The app has 1 module: *EngineSteer module*.

- For first evaluation of the module: read Big Picture, Engine Types, Designing the App, and Overview (with Simulation Scenarios).
- For integration planning: jump to System Architecture, SIM vs REAL, and Installation.
- For commissioning and trials: focus on Safety & Integration Path, Logs, and Watchdogs.
- For later reference: use the FAQ sections as a lookup, not a narrative.

The document is designed not to hide complexity or safety, but to deliberate them as they occur.

4. Big Picture

(a) No Engine Control, No USV

In every USV project, the pattern repeats.

The team rushes to integrate the "obvious hard parts" first – radar, AIS, cameras, links, remote start, clever OA logic.

You can have:

- the best OA algorithm,
- perfect camera detection on the horizon,
- beautifully engineered comms and power logic …

But sooner or later, the project manager discovers an uncomfortable truth:

- If the engines (the purchase always decided late and to be delivered even later) cannot be commanded safely and repeatably, there is no USV.
- Without reliable throttle and steering control, all the beautiful algorithms and sensor fusion remain stuck on the pier. The boat cannot sail the trial script, cannot follow the route, cannot demonstrate autonomy. The schedule slips, and suddenly the "engine control" piece becomes the critical path.

App_EngineSteer exists to make sure engine control is never again the reason your USV project stalls.

(b) Too Many Engine Styles, One Integrator Brain

Different engines and controllers speak different dialects:

- **Voltage-based** "lever" style (Yamaha-style, where gear is implied by voltage)
- **Percent-based** throttle with separate gear commands (typical control-head CAN style)
- **Discrete command** sets, safety / transfer rules, proprietary PGNs/message IDs
- **Discrete solenoid** sets, safety / transfer rules, proprietary Modbus message IDs

If your base GUI talks directly to each dialect, your code slowly turns into a fragile pile of special cases.

By the third boat, the base GUI is full of if (project == XYZ) branches and nobody remembers why.

(c) *App_Enginesteer* as your Single Abstraction Layer

The app hides the chaos and presents one unified model:

- **SET** vs **TRUE** for System, Throttle, and Steering
- Clear "gear / throttle / steering / active / transfer / idle" concepts

Your USV brain only ever speaks one JSON / TCP language; the app does the translation to whatever engine controller happens to be on that project.

USV brain → JSON → *App* → Engine dialect

(d) Three Built-In *Simulation* Scenarios

Even before real wiring, *App_EngineSteer* includes three built-in scenarios to support the full project life-cycle:

- **Random Simulation (Scenario 0)**:
    - Lively randomized moves in throttle and steering values that wander within safe limits, as well as show that the app is alive and your GUI can follow changing state.
    - Ideal for first-time UI bring-up, latency testing, and log recording.
- **Manual Slider Control / Training (Scenario 1)**:
    - The operator uses two sliders: **Throttle %** and **Steering %**. Large percentage readouts make it obvious what is being commanded.
    - Perfect for bench tests and dry-dock tuning without scripts.
    - Calm, predictable patterns for operator demos, UI tuning, and workflow rehearsals.
- **Real Loop Playback (Scenario 2)**:
    - Runs a controlled "real loop", designed to mirror how real engines would be exercised during bench or pier tests:
        - **Throttle** neutral → slow astern → full astern → back to neutral → slow forward → full forward → back to neutral.
        - Matching **steering** sweep from mid-ship → port limit → mid-ship → starboard limit → mid-ship.

    Designed for **repeatable demo runs** and log capture ("we did exactly the same pattern yesterday").

These scenarios are the sales engine for your project: they let you show motion, control, and repeatability long before the full autonomy stack is ready.

You don't need water, actuators, or even a full autonomy stack to convince a customer – the scenarios do that for you.

All three scenarios drive the **same** SET / TRUE panels, so whatever you debug in SIM will map cleanly when wired to REAL.

Scenario	Purpose	Typical Use
0 – Random	Health & liveliness	First-time hookup, latency demos
1 – Sliders	Manual play & training	Bench tests, UI / UX tuning
2 – Real loop	Repeatable ramp script	Bench / yard demonstrations, log capture

- **Flip SIM ↔ REAL** without rewriting your USV app

  In the future, when the CAN adapter and physical engine controller are ready, *Scenario 2* can be backed by real hardware instead of pure simulation.

  Your base application doesn't change its code or protocol — it still just talks to *App_EngineSteer*. Only the back-end wiring changes from "simulated engine" to "physical engine".

  The JSON contract is identical between SIM and REAL. Only the source of the numbers changes.

(e) Private Demo Videos on Request

Because the current wiring is not yet permanently installed on every bench, *App_EngineSteer* is offered initially with curated demo recordings:

- Live capture of the three scenarios
- CLEAR view of SET vs TRUE behaviour

These demo videos will show **use cases** installed for inboard, outboard, as well as waterjet engines.

If you want to see *App_EngineSteer* driving a real electronic control head, reach out for a private demo.

We can walk through the three scenarios, review the live logs together, and show exactly how your own USV base GUI can sit on top of the same engine layer with minimal effort.

If you're evaluating *App_EngineSteer*, email us and we'll send links to show:

- Scenario 0 – random sim
- Scenario 1 – slider training
- Scenario 2 – real loop (bench and yard versions)

(f) *App_Enginesteer* as the "Missing Piece"

This *white-label* app is a standalone **engine + steering control console** designed specifically for USV integrators:

- It speaks **multiple engine "dialects"** (voltage-based, duty-cycle, percent-based styles, and physical solenoids).
- It presents them to your base GUI as a single, unified control surface:
    - one throttle axis, one steering axis, and a simple TRUE/SET status layout.
- It is written as a *white-label* module – you do not have to show our screens to your customer; you can keep it running quietly behind your own GUI if you prefer.

The result is simple: you can **get the boat moving** long before you finish your "perfect" integrated UI.

(g) Most Integration Teams Under-Estimate Engine Control at the Start

For slightly larger engine types, talking to a modern engine drive system is not just "one more CAN message":

- Each vendor has **its own style** of expressing gear and power (voltage bands, percent ranges, duty cycles, "auto gear", etc.).
- The safety rules for neutral, transfer, local/remote, idle and emergency stop are **different for every project**.
- What looks like a "simple" throttle command can, if mishandled, cause an unsafe situation during sea trials.

App_EngineSteer does that hard, boring layer for you. It standardizes the chaos into:

- **SET** panels – what the USV **intends** to do (throttle % and steering %, per side if needed).
- **TRUE** panels – what the physical system is actually doing in real time.

Your base GUI talks in clean percentages and "remote/local" intent. *App_EngineSteer* handles the messy details underneath.

When something behaves strangely during a trial, SET/TRUE panels plus logs show exactly whether it was the USV brain, the adapter, or the engine controller.

(h) One Control Box, "Many Engine Styles"

Behind the simple sliders, *App_EngineSteer* understands multiple real-world engine control styles, for example:

- **Voltage-band gear logic**

  Classic electronic controls where the lever voltage decides gear (0-5V and/or 0-10V):

  - Below V_1 volts → REV (for 0-5V system, 2.3V → 0.5V [WOT])
  - Mid-band → NEUTRAL (for 0-5V system, 2.3V → 2.7V)
  - Above V_2 volts → FWD (for 0-5V system, 2.7V → 4.5V [WOT])
  - Gear is "implicit" inside the throttle signal.

  There is **no separate gear command**.

- **Pure throttle-percentage with explicit gear**

  Engines that expect:

  - A discrete NEU/FWD/REV command, and
  - A separate 0–100% throttle in that direction.

  Very common in CAN-based marine control heads.

- **Hybrid / PWM duty-cycle styles**

  Some systems express intent as PWM duty-cycle bands rather than pure voltage or percent.

  Internally, these are still mapped back to the same SET / TRUE % model.

App_EngineSteer hides these differences behind one **unified abstraction**.

Your base GUI never has to care **which** engine style is on this project – it only sees "Throttle%, Steering%, Gear (*automated*), Local/Remote, Idle, Active".

(i) Saving Your Team from Re-Inventing this Every Project

Without a module like *App_EngineSteer*, many integrators end up:

- Re-implementing engine control **from scratch** for every new boat.
- Embedding vendor-specific hacks deep inside their main GUI code.
- Debugging engine behaviour at the worst possible time – during sea trials.

With *App_EngineSteer*:

- Engine control becomes a **reusable building block**, not a one-off experiment.
- Your base GUI talks to a **stable, documented interface** instead of directly to hardware quirks.
- You can **simulate, record logs, and replay** motion patterns without touching the real engines until the shipyard is ready.

5. Engine Types

This section is **not** asking you to implement these – it shows how different engine styles all map back to the same *App_EngineSteer* interface.

Engine style → Adapter → App_EngineSteer SET/TRUE → JSON → Your USV

(a) Pure Mechanical Cable (Legacy but still Everywhere)

- **Signals**: push–pull cables for gear + throttle on the control head.
- **How it works:** helm lever moves → cable moves → throttle / gear on engine.
- **Integration pattern:** don't have to talk to the engine ECU at all – to move the cable or lever with a servo / linear actuator box.
- For integrator: this is just "mechanical adapter" on top of the SetGear / SetThrottle%.

App_EngineSteer stays the same; the actuator box turns commands into cable travel.

(b) Electro-Mechanical "Follow-Up" of Existing Head

- **Signals**:
    - Either one takes over the real control head (drive its lever with a motor), or
    - One has a second "remote head" that is mechanically slaved.
- **Integration pattern**:
    - The app outputs SetThrottle% / SetSteer%,
    - Local controller does the "follow-up" (move lever until feedback pot = target).

Again: this is just another **mapping layer** under *App_EngineSteer*; same Set/True logic.

As shipped today, this build runs in SIM-only mode; CAN/voltage adapters are attached per project.

(c) Analogue Throttle plus Separate Analogue Gear

Less common now, but it exists on some retrofits:

- One analogue channel: **0–5 V throttle** (no direction; just 0–100 %).
- Second or third channel: **discrete gear select** (e.g. 3-position switch or relay outputs for FWD/NEU/REV).

This is basically a halfway point between the two styles one already described:

- Treat as **digital gear + percentage** ("gear + %" family),

But the mapping layer turns that into *one* voltage and *some* relays.

So, you don't need a new abstraction – just a new *adapter*.

(d) CAN-Bus / J1939 with RPM-based Control

Some commercial engines don't really think in "throttle %" – they think in **desired RPM**:

- TX: DesiredEngineSpeed, GearCommand, "control source active" flags.
- RX: ActualEngineSpeed, ActualGear, a pile of status bits.

From your point of view this is only a **naming difference**:

- The app continues to use SetThrottle% (−100…+100).
- The mapping layer converts this to a desired RPM curve (idle → max).
- The True-panel shows either % or RPM, but mathematically it's the same range.

So, you can support "RPM engines" without changing any UI concepts.

(e) High-Level "Joystick / Vector" Systems

On pods, waterjets, or multi-screw DP systems you sometimes meet a controller that wants:

- Surge / sway / yaw demands (Fx, Fy, Mz) or
- "Ahead % / Sideways % / Rotate %" instead of raw engine/gear.

In that case, *App_EngineSteer* becomes the **low-level worker**, and a higher-layer box does:

- Convert joystick + DP logic into:
    - SetThrottlePort%, SetThrottleStbd%,
    - (plus thrust reverser / bucket position for jets),

Then the adapters convert to whatever the local engine style is (voltage, CAN, servo).

So again: no new engine *style* – just another level above you.

(f) Legacy Engine-Room Telegraph (For Completeness)

Old ships: "STOP / DEAD SLOW / SLOW / HALF / FULL / EMERGENCY" telegraph commands.

Modern USV work will almost never touch this directly; if it exists, someone will already have a **bridge control retrofit** that converts it into one of the styles above (usually CAN or analogue).

You don't need a zoo of special cases in your UI. Almost everything can be normalised to:

- **Set**:
    - SetGear (REV / NEU / FWD – if the hardware cares)
    - SetThrottle% (−100…+100, port & stbd)
    - SetSteer% (−100…+100)
- **True**:
    - TrueGear, TrueThrottle% or TrueRPM, TrueSteer%

Then each project plugs in its own **adapter**:

- Voltage curve with auto-gear (Yamaha style)
- Digital gear + % ("0–100 % both directions" style)
- RPM-centric J1939
- Cable/servo actuators, etc.

So yes, there *are* other flavours out there, but with the same abstraction:

one white-label control model, many thin hardware drivers …

6. Designing the App

(a) Dedicated Harness

It is a **dedicated engine + steering control harness** designed to sit between:

- The USV mission computer / autonomy stack, and
-
- Modern **digital engine & steering controllers** (twin engines, twin rudders, jets, etc.).

In its current form, *App_EngineSteer* runs in **full simulation mode**, exercising the exact logic you will later connect to the physical hardware.

Think of it as:

> A ready-made **engine room in a box** – throttle + steering brain included – just waiting for you to plug in the preferred control heads and CAN adapter.

(b) Before the Wires are Connected

Today, there is no physical throttle head or steering actuator connected.

By design. This gives three immediate benefits:

(1) **Safe, desk-bound validation**

You can prove concepts like:

- "Set vs True" behaviour (what you *ask* vs what the system *reports*).
- Command loops (AUTO remote control ↔ local bridge control).
- Scenario playback (ramp up / ramp down, gear transitions, etc.).

All without touching a live engine or risking hardware.

(2) **Clear separation of concerns**

- The **control logic and GUI** are already working:
  sliders, scenarios, "Set / True" views, system status, safety buttons.

- The **hardware binding** (actual CAN frames, voltages, OEM specifics) is a **thin layer to be completed per project**, not a rewrite.

(3) **A believable story for the end-customer**

- When you show *App_EngineSteer* in SIM mode, you are not hand-waving.

- You are showing a finished control workflow that only needs:

"Connect to the engine controller here."

(c) Running the Demo

When an integrator looks at *App_EngineSteer* today, the message is:

> **"All the *thinking* about how to drive engines and steering is already done. What remains is wiring to the specific hardware."**

Concretely, *App_EngineSteer* already provides:

- **Unified control view**
  - One console for steering, throttle, system status, and permissions (remote/local, active/inactive, idle/speed mode, etc.).
- **Set / True philosophy**
  - Every critical parameter is treated as:
    - **SET** – what the autonomy / operator is asking for.
    - **TRUE** – what the system reports is actually happening.

- **Scenario engine**
    - Randomised SIM runs,
    - Deterministic “real loop” ramps (−100% → 0% → +100% and back) for:
        - **Throttle,**
        - **Steering,**
        - **System state transitions (e.g., remote vs local).**

Even with no real CAN wire plugged in, an experienced integrator can immediately “see”:

- Where their command messages will enter,
- Where feedback from engine/steering will appear,
- How to hook their own USV supervisor into this control plane.

(d) Messaging Concept: 0xSYS / 0xTHR / 0xSTR

Internally the app groups CAN traffic into three conceptual channels:

- **0xSYS – System / Mode / Status**
  - System "health" / mode heartbeat.
  - Used as a **keep-alive** so the controller is comfortable while commands are changing.
  - Shown in the UI as **SET – SYSTEM – Status** (commanded) and **TRUE – SYSTEM – Status** (feedback).
- **0xTHR – Throttle**
  - Engine thrust command as a **percentage** (e.g. -100% = full astern, 0% = idle, +100% = full ahead).
  - Shown in the UI as:
    - **SET – THROTTLE – Cmd** (what App_EngineSteer is asking for).
    - **TRUE – THROTTLE – Status** (what the controller reports back).
- **0xSTR – Steering**
  - Steering demand as a **rudder percentage / angle** (e.g. -35%…0…+35%).
  - Shown in the UI as:
    - **SET – STEERING – Cmd**
    - **TRUE – STEERING – Status**

0xSYS / 0xTHR / 0xSTR are conceptual labels; the actual CAN IDs/wiring are adapter-defined per project.

(e) Neutral Referee

Every run is logged:

- Timestamp
- Direction (TX / RX)
- CAN ID (e.g. 0xSYS / 0xTHR / 0xSTR)
- 8 data bytes
- Human overlay (thr=+35%, rud=-12%, Scenario 2, Step 14)

When something misbehaves, you don't need arguments – you have a neutral referee log that shows exactly what happened.

(f) White-Label

It's white-label by design. No vendor names. No baked-in controller.

Just three concepts that work everywhere:

- SYSTEM
- THROTTLE
- STEERING

You can keep using it quietly behind your own brand.

(g) Safety & Integration Path

The harness is designed to be **commissioned safely**:

(1) **Bench / No Actuators**

- Kvaser device (e.g., Leaf v3) on the table, DEC powered, no hydraulics connected.
- Run Scenario 1 & 2 to confirm the DEC display and log outputs match expectations.

(2) **Yard / Actuators Connected, No Water**

- Connect hydraulic pumps and cylinders.
- Keep prop out of the water, use short scenarios and the **Emergency Stop** button on the GUI to prove that:
    - Commands can be started and stopped cleanly.
    - Steering follows expected angles with no runaway behaviour.

(3) **Sea Trials**

- Once shipyard is satisfied, the same EXE runs unchanged.
- Integrator can compare **on-water behaviour** against the original bench logs to confirm nothing in wiring or configuration has drifted.

This staged approach means **no surprises**: what the integrator sees in the video and logs is what they will see on their own bench.

At every phase, the same EXE runs unchanged.

These software safety paths complement – never replace – physical emergency stop and local control arrangements mandated by class and yard practice.

7. Overview



App_EngineSteer is a dedicated *white-label* control module responsible for **engine start/stop, throttle/gear control,** and **steering command execution** in both **SIM** and **REAL** environments.

It acts as the **deterministic bridge** between high-level navigation logic and low-level engine / steering hardware.

Unlike monolithic "all-in-one" control software, ***App_EngineSteer*** focuses **only** on propulsion and steering execution — making failures easy to isolate, recover, and debug.

(a) Core Responsibilities

- Execute **engine commands** (enable, start, stop, throttle, direction)
- Execute **steering commands** (rudder / hydraulic / actuator control)
- Enforce **safe command ranges** and neutral defaults
- Translate high-level intent into **hardware-specific outputs** (CAN / TCP / GPIO)
- Provide **SIM mode** for development, training, and dry-runs

(b) Reliability by Design

- Operates as an independent *enginesteer*-service process with its own runtime **heartbeat**,
- supports controlled **kill**, **restart**, **pause**, and **freeze** behaviours for operational testing and recovery,
- maintains observable operational state during degraded acquisition or transport conditions,
- and is designed to restart cleanly **without requiring restart** of downstream radar consumers or surrounding system layers.

If the *enginesteer* module encounters operational failure, it can be restarted independently **without** requiring restart of unrelated systems such as navigation, AIS, cameras, autonomy, or user-interface layers.

(c) SIM vs REAL Operation

(1) **SIM Mode**

- No hardware required
- Deterministic, repeatable behavior
- Ideal for integration, demos, and logic validation

(2) **REAL Mode**

- Direct control of engine and steering hardware
- Same command paths as SIM (no logic divergence)
- Hardware abstraction keeps upstream apps unchanged

(d) Why This Matters

By isolating engine and steering control into *App_EngineSteer*:

- Integrators avoid **cross-contaminated bugs**
- Navigation logic stays clean and testable
- Hardware changes do **not** ripple across the system
- Sea trials become predictable instead of stressful

(e) Intended Use

App_EngineSteer is designed to run **alongside** other *white-label* modules such as:

- Navigation / Autopilot
- GNSS / Boat motion
- AIS
- Radar
- Operator UI

Each module does **one job well**, and *App_EngineSteer* does **only what must be trusted most**.

8. Price

App_EngineSteer consists of a licensed control core and a commissioning phase, because no two boats are wired the same.

This *white-label* app is priced at **USD1,800** per core license per installation. Commissioning might take you **USD2,500**. For early adoptions, you would be given a **50% discount** [while stock lasts].

Depends on:

- Wiring complexity
- Engine type (outboard / waterjet / hydraulic)
- Number of actuators
- Sea trial involvement

The pricing of this *white-label* app is aligned with typical engineering component budgets for comparable navigation or control subsystems. It reflects the module's level of completeness, defined integration interface, and suitability for professional deployment. The objective is to offer a predictable, standalone component cost that can be evaluated alongside other system elements during project planning.

9. System Architecture

(a) High-Level Components

- **Command Ingress Service**

  Receives high-level engine and steering commands via TCP (JSON-based `GET / CMD / SET` style interface) from navigation logic, operator UI, or supervisory systems.

- **Engine Control Service**

  Manages engine enable, start/stop, throttle, and direction commands.

  Enforces neutral defaults, rate limits, and safe operating bounds before issuing hardware-level outputs.

- **Steering Control Service**

  Executes steering commands (rudder angle, hydraulic pump drive, or actuator position) with clamping, dead-zone handling, and return-to-neutral safeguards.

- **Hardware Abstraction Layer (HAL)**

  Translates logical engine and steering commands into hardware-specific interfaces (e.g. CAN bus, TCP, GPIO, or serial), allowing the same control logic to operate in both SIM and REAL modes.

- **Simulation Engine**

  Optional simulation layer that emulates engine response and steering behavior without physical hardware, enabling development, testing, and demonstrations.

- **Heartbeat & Watchdogs**

  Heartbeat writer and file-based watchdog mechanisms supporting health monitoring, stale detection, controlled shutdown, and automatic restart.

- **Configuration & Limits Manager**

  Loads vessel-specific configuration parameters (neutral points, limits, inversion flags, timing constants) and applies them consistently across all control paths.

(b) Unpacking

- Choose a drive, say <***d:***>.

- Create a sub-dir <***Apps***> (if you don't already have it), copy the zipped file (say, [***App_AIS_EngineSteer_Group.zip***]) into this sub-dir and do an [unzip] to unpack.

  - Dialog Msg – "*Opening these files might be harmful to your computer*", please click [OK].

  - Choose "*Extract All…*".

- Find and run [***App_EngineSteer.exe***] as **administrator**, choose the options, "*More info*" – "*Run anyway*". Please do allow the App to access the networks (both public and private), and it is now good to go.

- If necessary, do amend [***config.txt***] in the [***data***] directory for the correct parameters and do a restart of the App.

(c) Installation & Directories

- Application root, e.g.,
  <drive:>\Apps\App_EngineSteer_Group\App_ EngineSteer

- Data directory: …*App_EngineSteer\data* (auto-created)

- Log directory: …*App_EngineSteer\data*

- Config file: …*App_EngineSteer\data\config.txt*

- Splash Logo file:
  <drive:>\Apps\App_EngineSteer_Group\app_enginesteer_logo.png

- App EngineSteer Icon file:
  <drive:>\Apps\App_EngineSteer_Group\app_enginesteer.ico

This App sits on [*<drive>:\Apps\App_EngineSteer*]. In time, this main [*dir*] may be changed according to user's requirement.

It has a [*data*] dir,
[*<d:>\App_EngineSteer_Group\App_EngineSteer**__*data*__] where most of the native operations can be derived.

A sample of the [*data*] dir, known as [__*data.sample*__], is also enclosed for your reference.

Please see end section of this document for the full **file structure**.

10. Licensing & Trial Behaviour

On the first run [***config.txt***] is created, containing runtime settings plus a hardware-bound [**lock-id**].

A short trial window (for example 3 days) is granted automatically. After expiry, the app shows a clear message and exits cleanly.

The *white-label* app requires a simple activation step every 3 days. There is no time bomb, no crippled "demo features" – just a gentle unlock rhythm to remind you this is pre-license lab use, not yet a commercial deployment.

Deleting [*config.txt*] starts another short trial window or by reinstalling the zip package; this is intentionally inconvenient but not server-enforced. For long-term, installs an [**unlock-id**] (can be supplied and stored) in the [*config.txt*].

The *white-label* app is deliberately designed so integrators can experiment at zero cost.

11. Updates

From time to time, with newer features such as additional safety harnesses. You would be the first person to get this upgrade for free or at most at minimal upgrade cost with zero impact to your existing code base. Everything would be made genuinely **pin-compatible & backward-compatible**.

Please see [**Support**] section for actual details.

12. *config.txt*

In the [***data***] dir, there is a [***config.txt***] file, in here most of the configurations can be made and changed accordingly. It defines the following (sample):

```
# App_EngineSteer configuration (created automatically)
# Edit values and [restart app]. All times are in seconds unless stated.
# [comment lines] starts as [#] and must be on a separate line.

# EngineSteer START-UP #
Data_directory = Apps\App_EngineSteer_Group\App_EngineSteer\data

EngineSteer_command_port = 50201
EngineSteer_alt_command_port = 50202
EngineSteer_squawk_port = 50203

# EngineSteer CANBUS parameters #
EngineSteer_usb_port = xxxxx
EngineSteer_baudrate = xxxxx

# EngineSteer CANBUS Endian-ness # little or big #
can_endian = little
```

- EngineSteer local tcp ports are the ports that produce data feed:
    - EngineSteer_command_port = `50201` [implemented, on request]
    - EngineSteer_alt_command_port = `50202` [not yet implemented]
    - EngineSteer_squawk_port = `50203` [implemented, on request]

13. *Log_App_EngineSteer_xxx.txt*

The main [log] file is present to record and deduce any possible errors in the App.

- *[INFO] [App_EngineSteer] UI started.*
- *[WARN] [App_EngineSteer] Favicon not found at: D:\Apps\App_EngineSteer_Group\app_EngineSteer.ico*
- *[INFO] [App_EngineSteer] CAN mode = SIM (SimCanDriver).*

14. *..._heartbeat.txt*

The [*..._hb*] file is present to deduce if the App is still alive (nothing needs to be configured here !!!).

15. *kill_App_EngineSteer.txt*

You may try to click [close] or any possible closing buttons on the App, the App would be restarted again to continue operations.

If you try to close the App using the [**Task Manager**], you will only succeed about 50% of the time.

The App is indeed designed to be extremely difficult to be killed (stopped) using normal closing methods [**fault-tolerant**]. This is to ensure a reliable and working App during operations. Therefore, the only way to kill (stop) the App is via the presence of a [***kill_App_EngineSteer.txt***] in the [data] dir. When this [*kill_*] file is placed in the [*data*] dir, the App would be stopped.

This [*kill_*] file would be deleted automatically upon the next app run.

However, it will never spawn another similar copy. The App shall remain in singleton operation mode.

16. *restart_App_EngineSteer.txt*

The App would be restarted once there is the presence of a [***restart_App_ EngineSteer.txt***] in the [*data*] dir. When this file is present, the App will be restarted.

This [*restart_*] file would be deleted automatically upon the next app run.

17. *freeze_App_EngineSteer.txt*

The App would pretend-freeze its [*heartbeat*] once there is the presence of a [***freeze_App_EngineSteer.txt***] in the [*data*] dir. When this file is present, the App would freeze its heartbeat from beating. The App will then count-down on its stale-ness. Once the appropriate staleness level is reached, the App would do a [restart] itself.

The [*freeze_*] file would be automatically deleted upon the next app run.

18. Command Port Protocol (*not yet exposed in the demo app*)

The command port is line-oriented text over TCP. Commands and feedbacks are of the form: *GET / CMD / SET <param> <data>*. Responses are JSON terminated by CRLF.

(a) Core *GET / CMD / SET* commands

(1)	*get engine.info*	(includes both throttle(s)/steering(s) engine mode/info)
(2)	*set engine.mode <Real/Sim>*	(set engine mode)
(3)	*set throttle <…, …, …>*	(both engine throttles)
(4)	*set steering <…, …, …>*	(both rudder steerings)
(5)	*get health*	

Error responses: ERR unknown, ERR bad param, ERR bad data, ERR rate, ERR not found.

Using a port reader such as ***PuTTY*** App (copyrighted by *Simon Tatham*). Key-in the following detail:

- Host Name: localhost or **127.0.0.1**
- Port: **50201** (or your specified port address in the [*config.txt*])
- Connection Type: Other (Raw)

Click [Open].

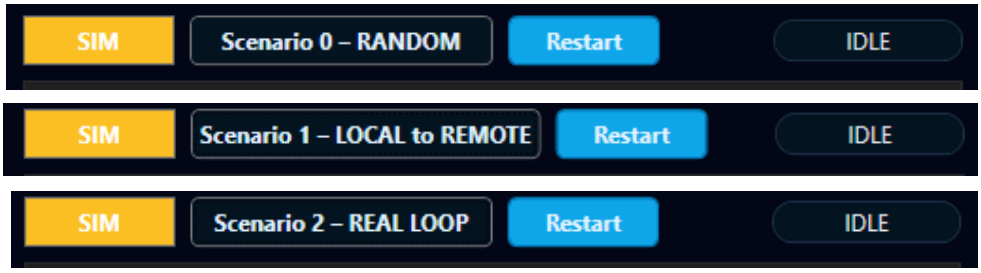
19. Squawk Port (not yet exposed in the demo app)

Using a port reader such as ***PuTTY*** App (copyrighted by *Simon Tatham*).
Key-in the following detail:

- Host Name: localhost or **`127.0.0.1`**
- Port: **`50203`** (or your specified port address)
- Connection Type: Other (Raw)

Click [Open].

20. Test Buttons



Near the bottom of the App window pane, the following buttons are displayed, starting from the left-hand side:

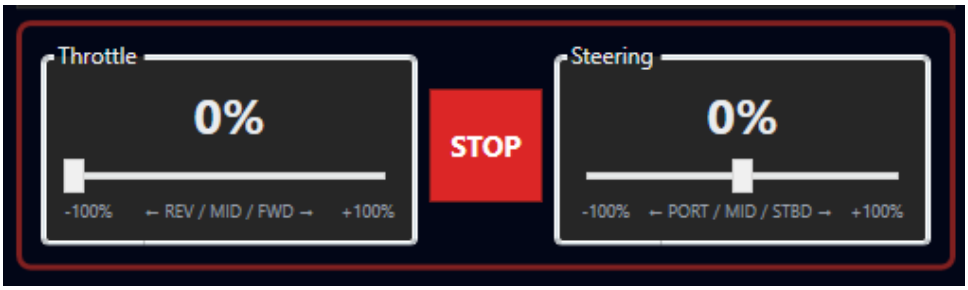
- [**REAL/SIM**] button – This button toggles between [Real] and [Sim] modes.

- [***Scenario NNN***] pill – This pill displays the current scenario being used when in [SIM] mode. This has no bearing for [REAL] mode.

- [**Restart**] button – This button conducts a clean restart of the App.

- [**IDLE/ALIVE/ALERT**] pill – This pill displays whether the engine/steering is active/idle or not. If in [ALERT] condition, the App would be recycled to ensure optimal operating condition.

21. Throttle and Steering Sliders



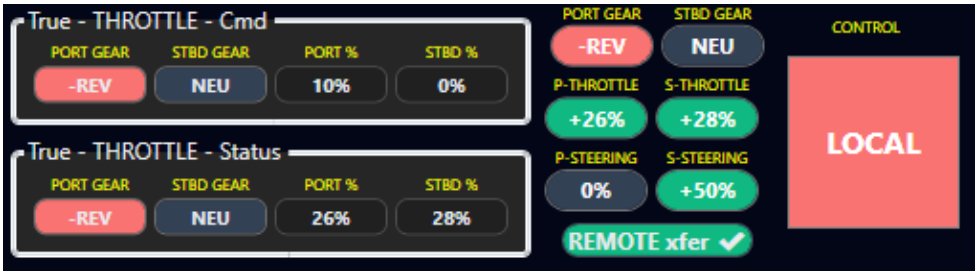
Near the top of the App window pane, the following sliders and button are displayed, starting from the left-hand side:

- [**Throttle**] slider – This slider provides manual control of the throttle from -100% to 0% to +100%. At -100%, this is the WOT in the reverse gear (no gear direction is needed to be inputted). At 0%, this is the neutral gear (no gear direction is needed to be inputted). At +100%, this is the WOT in the forward gear (no gear direction is needed to be inputted).

- [**(Emergency) Stop**] button – This button allows both the throttle and steering to be reset back to their zero positions (0% - zero throttle and 0% - zero steering).

- [**Steering**] slider – This slider provides manual control of the steering from -100% to 0% to +100%. At -100%, this is the 100% of the steering steered towards its port side (if its maximum port steering is limited at 35°, then, at this percentage point, its steering would be at 35° towards its port side). At 0%, this is the midships of the steering applied. At +100%, this is the 100% of the steering steered towards its stbd side (if its maximum stbd steering is limited at 35°, then, at this percentage point, its steering would be at 35° towards its stbd side).

22. True Status Reporting / Commanding



Just below the throttle and steering sliders, the following message pills and control button are displayed, starting from the left-hand side:

- [**True - THROTTLE - Cmd**] pills – These pills display the current true throttle command as received for its gears and percentages.

- [**True - THROTTLE - Status**] pills – These pills display the current true throttle status as reported for its gears and percentages.

- [**Miscellaneous**] pills – These pills display the gears, throttles, steerings status.

- [**REMOTE xter ?**] pill – This pill displays the current possibility of whether its local control could be transferred to remote control, as assessed by the miscellaneous pills.

- [**CONTROL**] button – As long as this button is not greyed out and the [**REMOTE xter ?**] pill is coloured green, this button can be clicked to transfer its local control to remote control, without any issue.

23. SET / True Status / Cmd Reporting / Commanding



Internally the app groups CAN traffic into three conceptual channels.

Just below the [CONTROL] button, the following message pills are displayed, starting from the left-hand side:

- **0xSYS – System / Mode / True Status / Set / Cmd**
    - System “health” / mode heartbeat.
    - Used as a **keep-alive** so the controller is comfortable while commands are changing.
    - Shown in the UI as **SET – SYSTEM – Status** (commanded) and **TRUE – SYSTEM – Status** (feedback).

- **0xTHR – Throttle True Status / Set / Cmd**
    - Engine thrust command as a **percentage** (e.g. -100% = full astern, 0% = idle, +100% = full ahead).
    - Shown in the UI as:
        - **SET – THROTTLE – Cmd** (what App_EngineSteer is asking for).
        - **TRUE – THROTTLE – Status** (what the controller reports back).
- **0xSTR – Steering True Status / Set / Cmd**
    - Steering demand as a **rudder percentage / angle** (e.g. -35%…0…+35%).
    - Shown in the UI as:
        - **SET – STEERING – Cmd**
        - **TRUE – STEERING – Status**

0xSYS / 0xTHR / 0xSTR are conceptual labels; the actual CAN IDs/wiring are adapter-defined per project.

24. Reliability & Watchdogs

The *App* cooperates with an external launcher through heartbeat files and markers:

- **Heartbeat** – periodically updated file indicating liveness.
- **kill**_* markers – trigger clean shutdown.
- **restart**_* markers – request a restart via the supervisor.
- **freeze**_ / **stale** markers – used for controlled stale-heartbeat behaviour.

These mechanisms are intentionally simple, externally observable, and fault-tolerant.

Currently, these operational controls are exposed primarily through the TCP command port and associated runtime supervision behaviour.

25. Known Limitations

- Each engine and steering has their own quirks. But we have tried several engine types to make this into something less reliant on particular engine behaviour.
- Command port does not implement authentication; deploy behind a firewall or on trusted networks.
- Behaviour described is for v1.03; the ***`GET / CMD / SET`*** **style protocol** is expected to remain stable.

26. Support

For support, changelog and feature requests, contact the engineer or integration partner who supplied this binary. When reporting issues, include version info, log excerpts and, if possible, a copy of the data (logs) directory.

You may replace the file [***app_enginesteer.ico***] (please see below) with any icon that represents your own brand. When the application runs, this icon will be used in the title bar and taskbar.

On a fully *unlocked* ***white-label App***, you are free to use any branding or message you wish. There is no copyright enforcement from the middleware provider.

(a) Upgrade Path

"If you want deeper integration, we will provide a dedicated custom adapter as a paid follow-up."

(b) Branding Favicon

To customize the application's favicon, simply replace the file:

```
<drive>:\Apps\App_EngineSteer_Group\app_enginesteer.ico
```

with any icon of your choice.

When the application starts, your custom icon will appear in the window title bar, taskbar, and any areas where the OS displays the application's icon.

On a fully unlocked version of the App, you are free to replace this favicon with any branding, logo, or identification you prefer.

This software is designed as a ***white-label middleware***, and therefore applies no restrictions or copyright enforcement on your chosen visual identity.

(c) Branding Splash Logo



To customize the startup splash screen, replace the file:

<drive>:\Apps\App_EngineSteer_Group\app_enginesteer_splash.png

with your preferred splash image (any resolution or aspect ratio is acceptable).

At runtime, the application will display your splash logo for approximately 0.5–2.0 seconds before the main interface loads.

On a fully unlocked version of the App, you are free to use any artwork, branding, message, disclaimer, copyright notice, or patent information you require.

The splash logo is best displayed when it is measured with width="750px" and height="375px" in size.

This software is intentionally provided as **white-label *middleware***, and your branding choices are fully unrestricted.

(d) Software Updates

The App is supplied with one year of software updates, performance improvements, and technical support.

Because engine/steering metadata formats, device behaviours, and regulatory requirements might evolve over time, maintaining full compatibility and reliability requires periodic updates.

Accordingly, continued updates, patches, and new feature releases are offered under an annual software maintenance plan.

This ensures that vessels always operate with the latest engine handling, metadata parsing, and stability improvements.

(e) Support Email

- Support is provided via email. Please write to:

  hello@contact.whitelabelusv.com

  or

  hello@whitelabelusv.com

  or

  white-labels@outlook.com

- Downloads are provided via website. Please go to:

  https://whitelabelusv.com/downloads

"We welcome collaboration with serious integrators, developers, researchers, and operational partners interested in advancing reliable and observable white-labelled systems. Such collaboration would be organised under a separate support and collaboration arrangement."

27. File Structure

It is recommended that you do follow the file structure as detailed below until you are more familiar with the interplay of the various ***white-label*** **Apps**:

```
<drive:>/
└──Apps/
    └──App_EngineSteer_Group/
        ├── app_enginesteer.ico
        ├── app_ enginesteer_logo.png
        └──App_EngineSteer/
            ├── App_EngineSteer.exe
            ├── …
            └── data/
                ├── sample.csv [not yet in use]
                ├── App_EngineSteer_heartbeat.txt
                ├── Log_App_EngineSteer_xxx.txt
                ├── config.txt
                ├── sim_enginesteer.txt [on request]
                ├── kill_*.txt
                ├── restart_*.txt
                ├── freeze_*.txt
                ├── test_pause_*.txt
                └── record_sim.txt [on request]
```

28. Compatibility

The App underwent comprehensive functional and scenario testing on both *Microsoft Windows 10* and *Windows 11* platforms. All test cases passed successfully, confirming full functionality and stability in these environments.

Microsoft Windows 10 and Windows 11 are copyrighted materials and proprietary software owned by Microsoft Corporation.

29. Intended Audience

This document is intended for system integrators and engineers. It is not an end-user documentation.

30. Errors & Omissions Excepted (E&OE)

This document reflects real integration work, observed behaviour, and verified configurations at the time of writing.

Hardware revisions, firmware versions, operating systems, drivers, and environmental conditions may introduce differences not covered here.

Integrators are expected to validate configurations within their own system context before deployment.

31. FAQs

(a) Need for Simulation?

Simulation lets integrators safely pre-play and snapshot almost all operating and failure scenarios — without cycling real hardware — so when rare edge cases appear at sea, they're recognized, rehearsed, and handled calmly using plans already drawn in simulation.

(b) Risk & Failure Handling?

(1) What happens if the *white-label* app stops responding or crashes?

Each *white-label* app includes a built-in **safety supervisor**. If the process becomes unresponsive, it can be externally killed or restarted without manual intervention. On restart, the app returns to a known safe state rather than resuming uncontrolled outputs.

(2) Can the *white-label* app be frozen without shutting it down?

Yes. A freeze command intentionally suspends all operational outputs while keeping the application alive, logging, and heart-beating. This allows safe diagnostics without losing visibility or requiring a restart.

(3) What happens if input data becomes stale or is lost?

All output paths are gated by input freshness checks. If valid input data is not received within a defined timeout, the application automatically blocks outputs to prevent acting on stale or misleading information.

(c) Integration & Time-to-Value?

(1) How long does it typically take to integrate a *white-label*?

Most integrations focus on wiring inputs and consuming clean JSON outputs. Because each *white-label* is self-contained and well-bounded, integration is typically measured in days rather than months.

(2) Can we test and validate the system before real hardware arrives?

Yes. Each *white-label* supports a simulation mode that mirrors real data paths. This allows full end-to-end testing, UI integration, and operator training before hardware is available.

(3) Do we need to change our existing system architecture?

No. Each *white-label* is designed to adapt to existing architectures rather than replace them. It exposes clear inputs and outputs and do not impose orchestration or control frameworks on the integrator.

(d) Control, Ownership & Lock-In?

(1) Do we retain control over deployment and operation?

Yes. Each *white-label* runs as a standalone application under your control. You decide how it is started, monitored, restarted, or integrated into your broader system.

(2) Can the *white-label* app be rebranded or embedded into our solution?

Yes. Each w*hite-label* is intended for rebranding and OEM use. Visual identity, naming, and external presentation can be adapted to match your product or platform.

(3) Are we locked into a proprietary ecosystem or roadmap?

No. Each *white-label* communicates through transparent, documented interfaces. You can integrate additional modules, replace components, or extend functionality without dependency on a closed ecosystem.

(e) Scope, Boundaries & Responsibilities?

(1) What does the *white-label* NOT do?

Each *white-label* focuses on a specific responsibility (e.g. AIS ingestion, engine control, GNSS input). They do not attempt to be a monolithic "all-in-one" system or replace higher-level decision logic.

(2) Where does our own logic fit in?

Your system remains the authority for mission logic, policies, and decision-making. Each *white-label* provides reliable data and controlled actuation but do not dictate operational behaviour.

(3) How are assumptions handled?

Operational assumptions and limits are explicit and documented. If an assumption is violated (e.g. missing data, invalid state), the *white-label* app fails safely rather than guessing.

(4) How does the app handles stress?

Each *white-label* is designed to be predictable under stress: clear boundaries, safe failure modes, and fast recovery — so integrators can focus on system behaviour.

(f) What is the best way to use a *white-label* like yours?

A group of *white-labels* forms an industrial stack. *White-labels* only make sense when the relationships between modules are deliberate.

True *white-label* modules do not assert identity, do not define the system narrative, do not compete for ownership, and do not attempt to be the "main product." Each module is intentionally insufficient as a standalone system, yet remains individually verifiable before integration.

Every *white-label* operates with a narrow contract, small API surface, predictable failure semantics, and minimal promises. This containment prevents scope creep, legal ambiguity, and support escalation.

In addition to operating on real devices, each *white-label* supports comprehensive **simulation** using the same data flows, timing, and failure behaviour. This allows integrators to develop, test, and validate systems long before physical hardware is available.

By using genuine *white-labels*, integrators retain branding, narrative control, and client ownership, while remaining free from attribution conflicts and copyright disputes.

(g) How does using your *white-labels* save time?

(1) Why not just build one big "all-in-one" control system?

Traditional seafaring already divides responsibility into specialists:

- Navigator — position, course, collision avoidance
- Radar operator — situational awareness and tracking
- Radio operator / AIS — identity and intent of nearby vessels
- Helmsman / Engine operator — propulsion and steering
- Lookout / cameras — visual confirmation

This separation exists because complexity is real.

Modern integrators did not inherit this division by accident — it evolved because no single interface or role can reliably handle everything at once.

Software that ignores this reality loses time later, even if it looks faster at the beginning. For example, many early integrators attempted to write a single app to handle all functionalities within a single unit, soon realized this one single app has become a debugging nightmare (because it collapses under real-world complexity) when hardware behaves differently per project, one subsystem misbehaves and pollutes the entire codebase, and no clear ownership exists over failures.

White-label middleware isolates complexity and responsibility. When something fails, only the fault domain needs examination. Nothing else is polluted.

White-labels are designed to replace human glue with deterministic software glue, which is where the time savings truly come from.

(2) How do *white-labels* actually save development time?

They eliminate **reinvention**. Each *white-label* already encapsulates protocol handling, device quirks, edge-case behaviour, and stable outputs. Without *white-labels*, teams repeatedly re-write the same drivers, re-discover the same bugs, and re-validate behaviour under pressure.

White-labels convert repeated effort into **one-time investment**.

(3) Why is modular middleware faster than custom integration?

Because **integration effort grows non-linearly**. Custom integration creates tight coupling, hidden dependencies, and cascading failures. *White-labels* expose **stable, minimal interfaces** (typically JSON or equivalent), allowing independent development, independent testing, and independent replacement.

Integration becomes assembly, not invention.

All *white-labels* are made backward-compatibility, so that earlier systems still work together with newer versions.

(4) How do *white-labels* help multi-team environments?

They prevent **knowledge silos**. In real teams, one engineer writes a subsystem, another debugs it months later, and context is lost. *White-labels* define clear ownership, localized logs, and predictable behaviour.

This dramatically reduces investigation and handover time.

(5) Isn't middleware an unnecessary extra layer?

Only once. After that every project starts from a working baseline, new engineers onboard faster, and systems scale without re-architecture.

Skipping middleware simply postpones complexity — it does not remove it.

(6) How do *white-labels* accelerate simulation and testing?

White-labels are designed to operate identically in simulation, replay, and live hardware. This enables early development without hardware, deterministic regression testing, and safe iteration without sea trials.

Simulation parity saves **months of calendar time**, not just coding hours.

(7) Why are standard data interfaces (e.g. JSON) important?

Because they are language-independent, version-tolerant, and human-readable. This ensures frontends and mission systems remain stable, internal refactoring does not break consumers, and long-term maintenance cost stays low.

Stable interfaces protect downstream systems from upstream change.

JSON is not slow in real systems because data rates are low and hardware latency dominates (as compared to UAV systems, which requires sub-second decisions, USV systems on the other hand, usually operate at nearer-to-second decisions). Its readability, stability, and compatibility save far more engineering time than any theoretical performance cost. JSON optimizes for reliability and integration speed, which are the real bottlenecks in operational systems.

Further, JSON is almost readable by human and hence, very easy to debug.

(8) How do *white-labels* reduce long-term maintenance effort?

They centralize expertise. Fixes, improvements, and optimizations are applied once, benefit all deployments, and do not fork across projects.

Without *white-labels*, every project becomes a permanent maintenance liability.

(9) How do *white-labels* speed up commercial delivery?

They allow stakeholders to see **working functionality immediately**. Instead of abstract promises, systems demonstrate live behaviour, outputs can be independently verified, and risk perception drops sharply.

Faster trust equals faster approvals and deployments.

(10) What is the biggest hidden time saving of *white-labels*?

Cognitive clarity. When systems are modular, engineers think locally, changes feel safe, and progress continues without fear.

White-labels don't just save engineering time — they preserve **momentum**.

White-labels do not save time by coding faster. They save time by preventing complexity from spreading. It is less wasted time overall.

(11) Why *white-labels* are critical before hardware arrives?

Hardware delays are inevitable.

By using JSON-based white-label modules, software development, simulation, and validation can proceed independently of hardware delivery.

When devices finally arrive, only final tuning remains — not architectural redesign.

(h) What happens if the *white-label* fails, hangs, or stops sending data?

Each *white-label* module outputs explicit heartbeat / status (e.g. alive, degraded, stale_data).

When heartbeat is missing or data stale, base system must fall back to its own safety policy (e.g. freeze commands, drop to manual, alarm), while the *white-label* module can be restarted as planned.

Each *white-label* design is "fail-silent with clear alarms", not "fail-mystery".

This separation ensures navigation/mission system always owns ultimate safety behaviour.

This is also explicitly discussed in FAQ (a).

(i) How do I switch between SIM and REAL modes, and are the outputs identical?

Each *white-label* module usually has a config setting (e.g. mode = SIM/REAL/REPLAY). AND/OR clickable button on the window pane. AND/OR remotely command-able TCP port(s).

JSON output structure is identical between modes; only the source of data changes. This allows: UI + autonomy code written once, tested in SIM, then moved to REAL with **no code changes**.

Logs from REAL can be replayed in REPLAY mode to debug issues without the boat.

(j) How do I see what the module is really doing internally?

Each *white-label* module writes rotating log files (/data/ModuleName_*YYYYMMDD*.txt).

Logs include input frames, decoded events, warnings, error stacks, and health status. JSON output can also be tapped and archived by the integrator for their own history / forensic analysis.

Strong logging means you don't need to guess — you can see.

(k) Can the *white-label* handle many channels / high update rates?

Each *white-label* module is designed for real maritime rates, not theoretical lab extremes.

Typical supported multiple input sources (e.g. dual GNSS, dual AIS, multiple cameras) with filtering and throttling.

JSON frequency / target density is configurable (e.g. limit to N updates/second or distance-based throttling).

If higher loads are needed, modules can be scaled horizontally (multiple instances per role).

(l) How do the *white-labels* connect to my system, and how is this kept secure?

Each *white-label* module talks via local TCP/UDP ports or shared JSON files, not internet/cloud.

Deployment is usually within a locked-down vessel LAN, under integrator's own firewall policies.

Recommended: run *white-labels* on dedicated VLAN / machine, and expose only the minimal ports to the main UI/mission system.

No secret remote backdoors; the integrator retains full network control.

(m) How do we handle updates without breaking an existing vessel or project?

Each *white-label* module has a semantic version (e.g. v1.2.3) and changelog.

JSON contract is backwards-compatible whenever possible (new fields added, old fields deprecated but not suddenly removed). Integrator controls when to upgrade; you can keep per-vessel module versions if needed.

Updates can be tested in SIM/REPLAY mode before touching live hardware.

(n) Can we extend the module or add our own fields/logic?

Core binary is closed / not modified, to keep it stable and supportable.

Extensibility is via:

- Config files (thresholds, filters, modes)
- Optional custom "sidecar" processes that consume JSON and add extra logic

If truly necessary, "custom builds" can be offered as a service under separate agreement.

This keeps the main line stable while still allowing project-specific behavior.

(o) What exactly are we allowed to do with the *white-label* module?

You are licensing the right to use the compiled module inside your solution, not the source code.

You may rebrand / reskin the UI and documentation (splash logo, favicon, colours, names) subject to basic attribution rules (if any).

You may sell systems that include the module; you do not resell the module standalone unless agreed.

The author retains core IP, which is why ongoing improvements can be supplied without re-negotiating every project.

This is also explicitly discussed in FAQ (c).

(p) How do multiple *white-label* modules work together without conflict?

Each *white-label* module exposes its output via well-defined ports / topics / namespaces (e.g. AIS, GNSS, Radar, Engine).

Your base system is free to:

- Fuse them (e.g. AIS+Radar track fusion)
- Cross-check (e.g. GNSS vs radar distance)

Modules do not secretly talk to each other; integration is fully owned by the base system. This keeps architecture clean and predictable, while still enabling a "system of systems".

This is also explicitly discussed in FAQ (j).

(q) How can we test and demo the system without an actual boat or sea trial?

Built-in SIM mode + pre-recorded log files allow realistic motion/tracks without hardware.

Integrators can:

- Demonstrate to customers
- Train crew / operators
- Validate mission logic

All using the same executable that will later run with real hardware, ensuring continuity.

This is also explicitly discussed in FAQ (b).

(r) Safe Use and Distribution (Multiple)

(1) Is it safe to run the downloaded application?

As with any downloaded executable, users may scan the package with any standard antivirus or security tool prior to execution.

(2) How is the application package distributed?

The application package is distributed via a secure, Cloudflare-hosted download link.

(3) Can I share or redistribute the application package?

Yes. Redistribution of the original, unmodified application package is permitted without prior notice or approval.

(4) May I transmit the package to third parties (e.g. integrators or evaluators)?

Yes. Users may transmit the original package to third parties for evaluation or integration purposes, provided the contents remain unmodified.

(5) Are modifications or repackaging allowed?

No. Redistribution is permitted only for the original, unmodified application package.

32. Document Change History

Document Revision and (Date)	Changes
B010 (2026-05-31)	• TCP Command Port • Corrected CANBus Timing Issues