white-label

middleware

App_Radar v1.04

Technical Integrator's Manual

combined with Practical Guide

For internal engineering and system integration use.
Not intended 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 1.04 D014 • Generated 2026-05-19

Content

Content 2

1. Quick Start Checklist 5

2. Private demo videos on request 6

3. How to read this document 7

4. Big Picture 8

(a) Problem Statement 8

(b) Radar As Echoes in the Dark 9

(c) Observing Clean Radar Screens versus Understanding Radar 17

(d) Environmental Reflection and Radar Visibility Reality 23

5. Designing the App 24

(a) Aim 24

(b) Acceptance of Radar Limitations 25

(c) Separate Radar Activity From Radar Reliability 26

(d) Multiple Radar Signals, One Operational Picture 27

(e) Radar Behaviour changes across Time 28

(f) Image Generation Is Not Radar Observation 29

(g) Transport Integrity Matters 30

(h) Replay-ability Matters 32

(i) Time is Treated as a First-Class Signal 33

(j) Designed for Degraded Radar Conditions, Not Just Normal Operation 34

(k) Integration Without Operational Assumptions 36

(l) Fault-Tolerant Operation and Radar Service Liveness 37

(m) Design Outcome 38

(n) Operational Visibility and Integration Philosophy 39

6. Overview 44

(a) Radar Operational Universe 44

(b) Parts of the Radar Module 46

(c) Core Responsibilities 50

(d) Reliability by Design 51

(e) SIM vs REAL Operation 51

(f) Why This Matters 52
(g) Intended Use 53
7. The Real Mode 54
(a) Typical Real Mode Workflow 55
(b) Summary 65
8. Price 66
9. System Architecture 67
(a) High-level components 67
(b) Unpacking 72
(c) Installation & Directories 73
10. Licensing & Trial Behaviour 75
11. Updates 76
12. *config.txt* 77
(a) General 77
(b) License 79
13. Command Port Protocol 80
(a) Core GET/CMD/SET commands 80
14. Reliability & Watchdogs 116
15. Known Limitations 117
16. Support 118
(a) Upgrade Path 118
(b) Branding Favicon 118
(c) Branding Splash Logo 119
(d) Software Updates 119
(e) Support Email 120
17. File Structure 121
18. Compatibility 122
19. Intended Audience 122
20. Errors & Omissions Excepted (E&OE) 122
21. The Math Behind the Radar Image Construction 123
(a) Preliminary Note 123
(b) Construction of the Radar Image 123

22. FAQs 132
(c) Need for Simulation? 132
(d) Risk & Failure Handling? 132
(e) Integration & Time-to-Value? 133
(f) Control, Ownership & Lock-In? 134
(g) Scope, Boundaries & Responsibilities? 135
(h) What is the best way to use a *white-label* like yours? 136
(i) How does using your *white-labels* save time? 137
(j) What happens if the *white-label* fails, hangs, or stops sending data? 141
(k) How do I switch between SIM and REAL modes, and are the outputs identical? 141
(l) How do I see what the module is really doing internally? 142
(m) Can the *white-label* handle many channels / high update rates? 142
(n) How do the *white-label*s connect to my system, and how is this kept secure? 142
(o) How do we handle updates without breaking an existing vessel or project? 143
(p) Can we extend the module or add our own fields/logic? 143
(q) What exactly are we allowed to do with the *white-label* module? 144
(r) How do multiple *white-label* modules work together without conflict? 144
(s) How can we test and demo the system without an actual boat or sea trial? 145
(t) Safe Use and Distribution (Multiple) 146

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_Radar.exe*** (as *administrator*) → [***config.txt***] is auto-created.
- At this point you can already run some simulations without any **Radar Antenna** 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.

CAUTION:

For subsequent startups, if the app fails to initialize or REAL Mode does not start successfully, the temporary evaluation license may have expired.

Telltale sign from logs:

2026-05-16 09:58:28.817 [license] validation failed: LICENSE_EXPIRED

To generate a fresh temporary evaluation license, delete the existing:

[<*drive:*>*/data/radar/data/****config.txt***]

file and restart the app.

A new temporary evaluation license will then be generated automatically during startup initialization.

Alternatively, please contact your vendor or deployment provider for a permanent long-term operational license.

2. Private demo videos on request

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

- Some radar images 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: *Radar module*.

- For first evaluation of the **Radar module**: read Big Picture, Designing the App, and Overview.
- For integration planning: jump to System Architecture, and Installation.
- For commissioning and trials: focus on Safety & Integration Path, Logs, and Watchdogs.
- For the more mathematically-inclined integrators, you can review the math behind the determination of the confidence envelopes.
- 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) Problem Statement

Marine radar systems sit at the foundation of modern situational awareness, navigation safety, collision avoidance, and autonomous vessel perception. Radar imagery, spoke data, target returns, and tracking information are consumed continuously by charting systems, ARPA logic, obstacle-avoidance systems, traffic analysis modules, and higher-level autonomy stacks. When radar data is correct, everything downstream appears stable, coherent, and trustworthy.

When radar data is wrong, the consequences can be significant.

Radar failures rarely announce themselves as obvious faults. Instead, they tend to degrade subtly — through stale sweeps, intermittent spokes, distorted returns, timing drift, incorrect scaling, ghost echoes, missing targets, multipath reflections, or imagery that appears visually plausible but is internally inconsistent. In many systems, these failures pass through silently, because confidence in radar imagery is usually assumed rather than actively evaluated.

Modern navigation systems are therefore vulnerable in a quiet way. Radar images may continue updating, overlays may continue aligning, sweep animations may continue rotating, and operators may see no explicit warning — yet the underlying situational truth may already have begun to diverge from reality.

This document starts from that simple reality. No blame is assigned, and no assumptions are made about intent or competence. The goal is only to acknowledge a fact: radar systems can fail in ways that are difficult to detect, and the consequences of undetected failure are often far more expensive than the failure itself.

(b) Radar As Echoes in the Dark

(1) Radar Never Truly "Sees" The World

One of the most dangerous misunderstandings in marine navigation is the belief that radar shows reality directly.

It does not.

Radar does not see ships.

It does not see coastlines.

It does not see danger.

Radar only transmits energy and listens for echoes returning from the darkness.

Everything displayed on a radar screen is therefore an interpretation of reflected energy across space and time.

This distinction matters enormously.

Because reflections are not truth.

They are clues.

And clues can mislead.

For most of maritime history, navigators understood this instinctively.

A lookout standing watch at night never assumed that every distant light meant safety, nor that every shadow represented danger. Experienced mariners knew visibility could deceive:

- fog could distort distance,
- rain could conceal motion,
- darkness could erase entire vessels,
- and reflections on water could create illusions.

Good navigators therefore developed discipline rather than certainty.

They:

- observed repeatedly,
- compared bearings over time,
- cross-checked with charts and motion,
- listened to surrounding traffic,
- and accepted that uncertainty was part of seamanship.

Situational awareness was never about perfect vision.

It was about managing incomplete truth.

(2) Radar Did Not Remove Uncertainty — It Automated Observation

Marine radar did not eliminate this uncertainty.

It automated it.

Instead of:

- human eyes scanning the horizon,

we now have:

- a rotating antenna transmitting pulses into darkness.

Instead of:

- human memory tracking movement,

we now have:

- digital sweep timing, spoke buffers, and target tracking software.

Instead of:

- manual plotting and visual estimation,

we now have:

- ARPA systems, filters, overlays, and prediction engines.

But the underlying problem never changed.

Radar still depends on:

- propagation conditions,
- geometry,
- timing accuracy,
- relative motion,
- environmental interference,
- and interpretation.

The sea did not become simpler because radar arrived.

The echoes merely became digital.

(3) The Dangerous Illusion Of "Seeing"

The most damaging consequence of modern radar systems is not technical.

It is psychological.

Because radar displays are:

- bright,
- smooth,
- stable,
- and visually convincing,

operators and downstream systems often begin treating radar imagery as reality itself.

This is a fundamental mistake.

A radar display is not a photograph of the world.

It is a continuously changing estimate constructed from delayed reflected energy.

The radar does not say:

"This object definitely exists exactly here."

It says:

"Something reflected energy in a way that currently resembles this interpretation."

Those are not the same statement.

And the difference becomes critical under degraded conditions.

(4) Echoes Can Lie Convincingly

Radar reflections are shaped by far more than object location alone.

Echoes are influenced by:

- target material,
- target angle,
- antenna height,
- sea state,
- rain,
- fog,
- multipath reflections,
- shoreline geometry,
- ducting conditions,
- and signal processing behaviour.

As a result:

- large targets can appear small,
- small targets can disappear entirely,
- ghost echoes can appear believable,
- shorelines can bloom outward,
- and clutter can masquerade as traffic.

A clean radar screen therefore does not guarantee understanding.

Sometimes the most dangerous radar picture is the one that looks calm, stable, and trustworthy while silently diverging from reality.

(5) Why Modern Radar Systems Fail Quietly

Traditional navigators confronted with uncertain visibility would naturally slow down.

They would:

- reduce speed,
- increase observation,
- rely on multiple information sources,
- and remain suspicious of incomplete information.

Modern systems often do the opposite.

Because radar imagery continues updating smoothly, systems frequently continue:

- tracking targets,
- projecting vectors,
- feeding collision-avoidance logic,
- and presenting confident overlays,

even when the underlying interpretation has become degraded or geometrically weak.

This is not because radar technology is poor.

It is because modern systems are often optimized to appear stable rather than to expose uncertainty honestly.

The system continues speaking confidently long after it should have admitted:

"I am no longer certain what I am looking at."

(6) Smoothing And Filtering Often Increase False Confidence

Modern radar systems include powerful processing layers such as:

- clutter suppression,
- target smoothing,
- image stabilization,
- auto-gain algorithms,
- automatic tracking filters,
- and predictive motion estimation.

These technologies improve usability significantly.

But they also introduce danger.

Why?

Because filtering can make degraded data appear visually trustworthy.

A target may continue moving smoothly across the screen even when:

- the underlying returns are intermittent,
- the spoke data is stale,
- the target geometry has weakened,
- or the tracking engine is merely coasting on prediction.

The human eye naturally trusts continuity.

That trust is exactly what can become dangerous.

A perfectly smooth radar trail may represent:

- old information,
- assumptions,
- or filtered hallucination rather than current truth.

(7) The Sea still Punishes Assumptions

The sea does not care:

- how expensive the radar is,
- how advanced the DSP algorithms are,
- how impressive the overlays appear,
- or how smooth the tracking looks.

It responds only to:

- actual objects,
- actual motion,
- actual timing,
- and actual decisions.

When systems mistake processed imagery for reality itself, they fail in the same way navigators fail when they trust a single misleading sighting without verification.

The difference today is speed.

Modern failures propagate faster.

Automation reacts faster.

And confident mistakes become dangerous earlier.

(8) Why this App Adopts a Stricter Philosophy

This app is designed around a deliberately conservative principle:

Radar is not truth.

Radar is evidence.

It assumes:

- all radar returns are conditional,
- confidence must remain observable,
- uncertainty must be exposed rather than hidden,
- and patience is sometimes safer than aggressive interpretation.

The app therefore does not attempt to make radar appear magically certain.

Instead, it treats radar exactly as a disciplined navigator would treat difficult visual observation:

- evaluate consistency,
- evaluate geometry,
- evaluate timing,
- reject implausible behaviour,
- slow down when confidence weakens,
- and acknowledge when truth becomes unclear.

(9) The Core Message

Radar did not solve situational awareness.

It automated observation.

But observation without judgment is merely fast confusion.

This app exists to make radar uncertainty visible again — not to weaken navigation, but to make it honest.

The echoes still govern what the vessel believes about the surrounding world.

They simply arrive now as packets, sweeps, spokes, and processed imagery.

And like all measurements, they must be interpreted carefully — not worshipped blindly.

"Radar does not show reality directly.
It shows reflected energy interpreted through software — and interpretation without judgment is how confident mistakes become expensive."

(c) Observing Clean Radar Screens versus Understanding Radar

(1) The Common Assumption

A widely held belief in marine integration goes like this:

- a small radar gives weak awareness,
- a large radar gives strong awareness,
- open-array systems outperform compact domes,
- commercial-grade hardware must produce better truth,
- and therefore: spending more money naturally produces safer navigation.

This belief is understandable — but incomplete.

Modern radar systems are extremely capable.

Higher-end hardware absolutely improves:

- detection range,
- receiver sensitivity,
- target separation,
- clutter suppression,
- and environmental robustness.

These improvements are real and important.

But radar systems do not directly measure truth.

They measure reflected energy.

And reflected energy can be misleading.

(2) Why Expensive Radar Systems Still Produce Misleading Situations

Radar performance depends on far more than hardware quality alone.

Radar behaviour is continuously affected by:

- sea state,
- rain intensity,
- target material,
- target angle,
- shoreline reflections,
- antenna motion,
- propagation conditions,
- and surrounding geometry.

As a result:

- large targets may weaken unexpectedly,
- small fiberglass vessels may disappear intermittently,
- clutter may resemble moving traffic,
- and ghost echoes may appear visually convincing.

Even under good operating conditions, radar systems still require interpretation.

A clean radar display does not automatically mean the surrounding situation is fully understood.

(3) The False Leap: Stable Display ≠ Correct Interpretation

One of the most common integration mistakes is confusing:

"*The radar picture looks stable*"

with

"*The situational interpretation is correct.*"

Those are not equivalent statements.

Modern radar systems are designed to produce:

- smooth sweeps,
- stable targets,
- clean overlays,
- and continuous tracking behaviour.

These features improve usability significantly.

However, visual stability can sometimes conceal uncertainty.

Targets may continue appearing smooth and believable even when:

- return strength is degrading,
- spoke consistency is weakening,
- tracking filters are coasting,
- or timing alignment is no longer ideal.

The display remains visually calm even while confidence should be decreasing.

(4) Why Tracking Systems Can Increase False Confidence

Modern radar tracking systems such as ARPA are extremely useful operational tools.

But they also introduce an important psychological effect:

once a target receives:

- a vector,
- a trail,
- a CPA marker,
- or a stable prediction line,

operators naturally begin treating that target as “understood.”

In reality, many tracking systems continue extrapolating motion smoothly even when underlying radar evidence becomes intermittent or degraded.

This is not necessarily a fault in the radar.

It is a natural consequence of predictive filtering.

The danger arises when filtered continuity is mistaken for confirmed truth.

(5) The Real Problem Is Usually Interpretation, Not Detection

Modern radar hardware is rarely the primary limitation.

The more difficult problems usually involve:

- target interpretation,
- timing behaviour,
- stale-data handling,
- tracking assumptions,
- overlay synchronization,
- and downstream software decisions.

These behaviours are often invisible during normal operation.

They usually become apparent only under stress, cluttered environments, degraded visibility, or complex traffic situations.

This is why expensive hardware alone cannot guarantee reliable situational awareness.

(6) Why The Belief Persists

This belief survives because radar systems are visually persuasive.

Bright displays, stable tracks, and smooth overlays naturally create confidence.

And most radar failures are not dramatic.

The most dangerous failures are often subtle:

- a delayed track,
- an over-smoothed target,
- a fading vessel,
- a stale overlay,
- or a believable but incorrect interpretation.

Because the display continues appearing professional and stable, the degradation may remain unnoticed until operational consequences appear.

(7) What Actually Improves Radar Trustworthiness

What does NOT solve the problem:

- more transmit power alone,
- larger antennas alone,
- more aggressive filtering,
- or smoother graphics.

What DOES improve trustworthiness:

- visible confidence degradation,
- explicit stale-data detection,
- sweep consistency validation,
- honest uncertainty exposure,
- deterministic replay of failures,
- and systems willing to question their own interpretation.

This app is designed around exactly that philosophy.

It does not assume that visually convincing radar behaviour automatically represents reliable situational understanding.

Instead, it continuously evaluates whether the displayed behaviour still remains internally consistent and operationally believable.

(8) The Engineering Reality

Radar is one of the most powerful tools in marine navigation.

But radar has never eliminated uncertainty.

It has only transformed uncertainty into processed echoes, tracked motion, and interpreted imagery.

The important engineering distinction is therefore not:

"Does the radar display look convincing?"

The more important question is:

"Does the system still understand the limits of what it currently knows?"

This app exists to keep that distinction visible.

"Better radar hardware improves observation.
Only disciplined interpretation prevents confident mistakes."

(d) Environmental Reflection and Radar Visibility Reality

All marine radar observation ultimately depends on how transmitted energy interacts with the surrounding environment.

At any given moment, sea state, rainfall, target material, shoreline geometry, antenna motion, atmospheric conditions, and surrounding traffic collectively determine what radar can meaningfully observe, separate, and track. These conditions are never fixed. They evolve continuously as the vessel moves, waves change shape, weather shifts, targets alter aspect angles, and reflections behave differently across time.

As a result, radar visibility is always conditional.

Even with a fully functional radar system and strong signal returns, situational understanding remains constrained by current environmental behaviour. Certain conditions amplify clutter, distort target shape, weaken returns, create ghost echoes, or reduce separation between genuine traffic and reflected artefacts. These conditions may persist briefly or for extended periods, but during that time the operational meaning of the radar picture can change significantly.

No antenna size, firmware revision, or signal-processing chain can fully override this reality.

No amount of downstream filtering can recover information that was never cleanly observable in the first place. Once reflections become ambiguous, intermittent, masked, or geometrically distorted, software can only attempt interpretation — it cannot recreate certainty that the environment never provided.

The implication is unavoidable.

Radar interpretation cannot be declared reliable once and trusted indefinitely. It must be evaluated continuously and operationally as a function of live environmental conditions, return behaviour, sweep consistency, target continuity, and observable radar context.

Anything else is not situational awareness — it is assumption.

5. Designing the App

(a) Aim

App_Radar aimed to be a fault-tolerant radar acquisition and radar-service module for marine radar systems. Its role is to reliably acquire, process, generate, transport, and expose radar spoke and radar image data in a transparent and operationally observable manner.

The app interfaces directly with supported radar hardware and continuously handles:

- radar spokes,
- sweep timing,
- antenna state,
- radar configuration,
- image generation,
- radar mode transitions,
- and radar health behaviour.

Collectively, these are referred to as the "***Radar State Variables***."

In simulation mode, the app additionally supports deterministic radar replay, controlled spoke generation, and repeatable development testing under both nominal and degraded operating conditions.

The design philosophy is intentionally operational rather than cosmetic.

Radar imagery may continue appearing visually stable even while underlying spoke continuity, sweep freshness, transport timing, or radar acquisition behaviour are degrading silently. For this reason, the app does not treat the mere existence of radar imagery as proof of healthy radar operation.

Instead, the app continuously monitors operational characteristics such as:

- sweep continuity,
- spoke freshness,
- image update timing,
- radar state transitions,
- data liveness,
- and transport integrity.

Collectively, these are referred to as the "***Radar Operational Variables***."

The goal is therefore not merely to display radar imagery.

The goal is to expose radar behaviour predictably and transparently so that downstream systems, integrators, and operators can reason about radar availability, freshness, and operational state directly.

This app is designed as a radar foundation layer rather than a black-box display engine.

This section describes the design principles that shaped the ***radar-head module*** of the app.

(b) Acceptance of Radar Limitations

The first design decision is operational rather than graphical:

Radar is reflective observation, not direct vision.

A radar system does not observe the surrounding world directly. It transmits energy and processes returning reflections generated by the environment. The resulting radar imagery is therefore shaped not only by nearby objects, but also by sea state, rainfall, target material, shoreline reflections, antenna motion, spoke timing, and signal-processing behaviour.

As a result, radar imagery is always conditional.

A visually stable radar screen does not automatically imply stable radar acquisition.

Strong echoes do not always represent strong situational understanding.

Clean imagery does not guarantee clean observation conditions.

For this reason, the app avoids treating radar output as self-validating.

Instead, radar behaviour is continuously monitored through operational characteristics such as "***Radar Operational Variables***."

The objective is not merely to generate radar imagery.

The objective is to expose whether the radar system is still operating in a timely, coherent, and operationally consistent manner.

(c) Separate Radar Activity from Radar Reliability

A rotating antenna and an updating radar screen do not necessarily indicate healthy radar acquisition.

Many radar systems continue appearing operational even while underlying acquisition quality is degrading silently.

For example:

- the antenna may continue rotating while spoke delivery stalls,
- radar imagery may continue refreshing while sweep freshness drifts,
- image buffers may continue displaying older spokes,
- or transport delays may accumulate while the display still appears visually smooth.

These conditions are often difficult to notice during normal operation because the radar screen itself may continue looking stable and believable.

For this reason, the app explicitly separates:

- radar activity — whether the system is still producing observable output,

from

- radar reliability — whether the underlying radar acquisition behaviour remains timely, fresh, and operationally consistent.

The app therefore does not treat visible radar imagery as automatic proof of healthy radar operation.

Instead, operational behaviour itself remains continuously observable through characteristics such as "***Radar Operational Variables***."

This separation helps expose subtle acquisition problems earlier, before stale or degraded radar behaviour silently propagates downstream.

(d) Multiple Radar Signals, One Operational Picture

A marine radar system does not produce a single piece of information.

It continuously generates multiple operational signals in parallel, including "***Radar State Variables***."

These signals are closely related and must remain operationally consistent with one another.

For example:

- spoke arrival behaviour should align with antenna rotation,
- image updates should reflect current sweep activity,
- transmit state should correspond with observable radar output,
- and radar mode transitions should occur coherently across the acquisition pipeline.

When these operational signals begin diverging, subtle radar degradation can occur even while the radar display still appears visually stable.

Examples include:

- imagery continuing while spoke freshness degrades,
- partial sweeps appearing during transport interruptions,
- stale image buffers surviving range transitions,
- or antenna rotation continuing while acquisition stalls.

For this reason, the app treats radar operation as a coordinated operational picture rather than as isolated data streams.

The goal is not merely to confirm that "something is happening."

The goal is to verify that radar acquisition, sweep generation, image production, and radar state behaviour remain operationally aligned and internally consistent across time.

(e) Radar Behaviour changes across Time

Radar acquisition is inherently time-dependent.

A radar system is not producing a single static image. It is continuously building radar imagery progressively through rotating sweeps, spoke acquisition, image buffering, and transport timing.

As a result, radar behaviour can degrade gradually across time even while the display itself continues appearing visually stable.

For example:

- spokes may begin arriving irregularly,
- sweep timing may drift under transport load,
- image freshness may slowly increase,
- buffer latency may accumulate,
- or stale spokes may persist briefly inside generated imagery.

These conditions are often subtle during early degradation stages.

The radar picture may still look smooth and believable even while operational timing behaviour is no longer ideal.

For this reason, freshness and timing behaviour are treated as operational characteristics rather than secondary diagnostics.

The app therefore continuously monitors behaviours such as "***Radar Operational Variables***."

The objective is not simply to determine whether radar data exists.

The objective is to determine whether the radar system is still operating in a timely and operationally coherent manner as conditions evolve across time.

(f) Image Generation is Not Radar Observation

Radar imagery is a processed representation of radar acquisition behaviour.

It is not the raw observation itself.

Modern radar displays are typically constructed from:

- spoke accumulation,
- image buffering,
- sweep interpolation,
- persistence effects,
- scaling,
- and graphical rendering pipelines.

These processes improve readability and visual continuity significantly.

However, they can also conceal early operational degradation.

For example:

- stale spokes may remain briefly visible inside image buffers,
- partial sweeps may appear visually complete,
- interpolated imagery may smooth over missing acquisition gaps,
- or delayed transport behaviour may still produce believable radar pictures.

As a result, a visually stable radar image does not necessarily imply fresh or healthy radar acquisition underneath.

This distinction is important operationally.

A downstream system observing only generated imagery may not immediately detect:

- spoke delivery interruptions,
- timing drift,
- transport jitter,
- or stale acquisition behaviour.

For this reason, the app treats image generation and radar acquisition as related but separate operational layers.

The app therefore monitors not only the generated radar imagery itself, but also the underlying acquisition behaviour responsible for producing it.

The objective is to prevent processed imagery from unintentionally masking gradual degradation in the underlying radar pipeline.

(g) Transport Integrity Matters

Radar data transport is a real-time operational problem rather than a simple networking problem.

Marine radar systems continuously generate large volumes of spoke and sweep data that must be transported, processed, buffered, and rendered within tight operational timing constraints.

Under degraded conditions, transport behaviour itself can become a source of radar degradation.

Examples include:

- UDP packet loss,
- burst spoke arrival,
- transport jitter,
- buffer accumulation,
- stale image persistence,
- or delayed sweep delivery under system load.

These conditions do not always produce immediate visible failures.

In many cases, radar imagery may continue appearing visually stable while transport timing behaviour has already begun drifting operationally.

For example:

- spokes may arrive unevenly across sweeps,
- image latency may gradually increase,
- or stale buffers may survive longer than intended during transport interruptions.

Because of this, the app treats transport behaviour as part of radar operation itself rather than as a separate infrastructure concern.

The app therefore continuously observes characteristics such as:

- spoke delivery timing,
- data liveness,
- buffer freshness,
- sweep continuity,
- and transport stability under changing load conditions.

The objective is not merely to keep radar data flowing.

The objective is to ensure that transported radar behaviour remains timely, coherent, and operationally representative of the actual radar state.

(h) Replay-ability Matters

Many radar acquisition problems are transient, timing-dependent, and difficult to reproduce consistently.

Examples include:

- intermittent spoke loss,
- transport jitter,
- delayed sweep delivery,
- stale image persistence,
- radar mode transition timing,
- or brief acquisition interruptions under changing environmental or system conditions.

These behaviours may appear only briefly during live operation and may disappear before meaningful diagnosis can occur.

As a result, relying solely on live observation often makes troubleshooting incomplete, inconsistent, or highly dependent on timing.

For this reason, the app places strong emphasis on deterministic replay capability.

Replay-ability allows previously captured radar behaviour to be:

- repeated consistently,
- observed under controlled conditions,
- compared across software revisions,
- and analysed without requiring the original live event to recur.

This is particularly important for radar systems because many operational problems are not permanent failures.

They are timing-sensitive behaviours that emerge only under specific combinations of:

- transport load,
- sweep timing,
- system latency,
- radar state transitions,
- and acquisition conditions.

The ability to replay radar behaviour deterministically therefore becomes an important engineering and diagnostic capability rather than merely a simulation feature.

The objective is not simply to reproduce radar imagery.

The objective is to reproduce operational radar behaviour closely enough that subtle acquisition and transport problems can be analysed, tested, and improved systematically.

(i) Time is Treated as a First-Class Signal

Time behaviour is a critical part of radar operation.

Radar acquisition depends heavily on timing consistency across spoke generation, sweep progression, image updates, transport delivery, and downstream processing.

For this reason, time is treated as a primary operational characteristic rather than a background utility.

The app continuously observes behaviours such as "***Radar Operational Variables***."

A timing disturbance does not always produce an immediate visible radar failure.

For example:

- spokes may continue arriving while latency accumulates,
- sweeps may remain visually smooth while freshness drifts,
- or buffered imagery may continue updating while underlying acquisition timing becomes increasingly delayed.

These behaviours can quietly distort operational radar behaviour even though the display itself still appears visually stable.

For this reason, the app treats timing integrity with the same operational importance as spoke acquisition itself.

This is particularly important in systems where downstream consumers implicitly assume:

- fresh radar imagery,
- continuous sweep progression,
- and operationally current radar behaviour.

The objective is not merely to ensure that radar data exists.

The objective is to ensure that radar behaviour remains temporally coherent across the entire acquisition and transport pipeline.

(j) Designed for Degraded Radar Conditions, not just Normal Operation

Many radar applications are designed primarily around nominal radar behaviour. Degradation handling is often treated as secondary or reactive.

This app is designed with degraded radar operation as an expected operational condition.

Scenarios such as:

- partial spoke loss,
- stale sweep persistence,
- transport jitter,
- delayed image updates,
- radar mode transition interruptions,
- buffer accumulation,
- and intermittent acquisition stalls

are not treated as rare edge cases.

They are treated as normal engineering conditions that must remain observable and diagnosable during operation.

As a result, the app is designed to continue exposing meaningful operational behaviour even while radar acquisition quality degrades.

The objective is not merely to maintain a visually stable radar display.

The objective is to ensure that degradation remains visible, understandable, and operationally traceable rather than silently hidden underneath believable radar imagery.

This becomes particularly important during transient or timing-sensitive failures where radar behaviour may degrade gradually rather than fail completely.

(k) Integration Without Operational Assumptions

The app is intentionally conservative in what it assumes about downstream integration.

It does not assume:

- how radar imagery will be consumed,
- what downstream processing exists,
- whether tracking systems are present,
- whether overlays are synchronized externally,
- or whether human supervision is continuously available.

Instead, the app focuses on exposing radar acquisition and operational behaviour as transparently as possible.

This includes observable characteristics such as "***Radar Operational Variables***."

By exposing operational behaviour directly rather than hiding it behind simplified status assumptions, integrators can make their own decisions regarding:

- redundancy handling,
- stale-data tolerance,
- fallback behaviour,
- replay analysis,
- and downstream operational gating.

This design allows the app to function as a foundational radar service layer rather than as a tightly coupled black-box radar display system.

(l) Fault-Tolerant Operation and Radar Service Liveness

In addition to handling radar acquisition, the app is designed to continuously monitor its own operational liveness.

Real-world radar deployments must assume that:

- processes may stall,
- threads may deadlock,
- I/O may block,
- transport pipelines may freeze,
- and long-running systems may degrade unpredictably across time.

In many situations, silent degradation is more operationally dangerous than an explicit crash because downstream systems may continue consuming stale radar behaviour without realizing that acquisition is no longer progressing correctly.

For this reason, the app implements explicit fault-tolerant operational mechanisms, including:

- **heartbeat** monitoring to verify that acquisition and processing loops remain active,

- **stale detection** to identify radar behaviour that appears present but is no longer updating meaningfully,

- **pause-state** handling to distinguish intentional suspension from unintended acquisition stoppage,

- controlled **kill** mechanisms for deterministic shutdown under unrecoverable conditions,

- and **automatic restart** behaviour to recover from transient operational failures without requiring manual intervention.

These mechanisms are intentionally simple, externally observable, and operationally transparent.

They are not designed to hide failures.

They are designed to expose failures early, bound their operational impact, and support predictable recovery behaviour.

The design philosophy is straightforward:

A radar service that can clearly expose degraded operation is safer than one that continues presenting believable radar behaviour while underlying acquisition has stalled silently.

By treating software liveness as part of radar operation itself rather than as an external assumption, the app remains operationally observable not only during nominal acquisition, but also during degraded or recovering states.

(m) Design Outcome

The result of these design choices is not a more visually impressive radar display, nor a more aggressively processed radar image.

It is a radar acquisition and radar-service layer that:

- exposes operational radar behaviour transparently,
- surfaces degradation early,
- keeps sweep and spoke behaviour observable,
- maintains operational liveness visibility,
- and remains fault-tolerant under degraded conditions.

The app does not attempt to hide radar acquisition problems behind visually stable imagery.

Instead, it is designed to make degraded radar behaviour operationally visible before it silently propagates downstream.

This includes conditions such as:

- stale spokes,
- sweep interruptions,
- transport jitter,
- delayed image updates,
- acquisition stalls,
- and timing drift across the radar pipeline.

The objective is not merely to keep radar imagery visible.

The objective is to ensure that downstream systems, integrators, and operators can continue reasoning about radar freshness, radar availability, and acquisition behaviour explicitly as operating conditions evolve.

In practice, this distinction becomes important long before complete radar failure occurs.

(n) Operational Visibility and Integration Philosophy

Naturally, this leads to the app exposing operational radar characteristics such as "***Radar Operational Variables***."

These are not intended as marketing metrics or artificial "health scores."

They are operational indicators intended to help integrators and operators reason about current radar acquisition behaviour directly.

This design avoids two common integration problems:

- exposing only low-level diagnostics that are difficult to interpret operationally,

and

- hiding degraded radar behaviour behind visually convincing imagery.

Operational states and timing indicators are intentionally simple because:

- engineers can troubleshoot them,
- operators can monitor them,
- and integrators can propagate them into their own systems without requiring proprietary interpretation logic.

The objective is operational clarity rather than cosmetic reassurance.

(1) Why degraded scenarios are treated as design inputs

Radar acquisition problems are often intermittent, timing-dependent, and difficult to reproduce consistently.

Examples include:

- partial spoke loss,
- stale image persistence,
- sweep interruptions,
- transport jitter,
- delayed image updates,
- and transient acquisition stalls.

These conditions are not treated as rare edge cases.

They are treated as expected operational behaviours that the system must continue exposing clearly and predictably.

This allows degradation behaviour to remain diagnosable even when radar imagery itself still appears visually stable.

(2) Why operational scenarios are explicitly enumerated

The operational scenarios documented within the system are not merely test cases.

They are intended to capture specific radar acquisition and transport behaviours under controlled conditions.

Each scenario helps answer practical engineering questions such as:

- Is spoke acquisition progressing correctly?
- Are sweeps remaining fresh?
- Is transport timing behaving consistently?
- Do image updates still reflect live radar behaviour?
- Does the system remain operationally observable under degraded conditions?

By explicitly documenting and replaying these conditions:

- degradation behaviour becomes discussable,
- regressions become easier to detect,
- and operational behaviour becomes easier to reproduce consistently across software revisions.

(3) Why the system avoids vendor-specific operational assumptions

This app does not attempt to declare one radar vendor, antenna type, or firmware family universally superior.

Different radar systems behave differently under different:

- installation environments,
- transport architectures,
- sea conditions,
- and operational loads.

For this reason, the app focuses primarily on exposing observable operational behaviour rather than embedding rigid assumptions about how a particular radar implementation “should” behave internally.

This allows the radar service layer to remain adaptable across changing hardware generations and deployment environments.

(4) Why simulation and replay are treated seriously

Simulation and deterministic replay are treated as operational engineering tools rather than secondary features.

Radar acquisition problems are often difficult to reproduce live because they may depend on:

- timing,
- system load,
- transport behaviour,
- or transient environmental conditions.

Replay capability allows these behaviours to be:

- repeated consistently,
- analysed under controlled conditions,
- and compared across software revisions.

This supports more reliable troubleshooting, testing, and long-term operational improvement.

(5) What this system intentionally does not try to do

To remain operationally transparent, the app intentionally avoids:

- hiding stale acquisition behaviour,
- masking degraded timing conditions behind smoothing,
- presenting visually stable imagery as automatic proof of healthy operation,
- or tightly coupling radar acquisition with downstream interpretation logic.

These are intentional design boundaries rather than missing features.

The app is designed to function as an observable radar acquisition and radar-service layer whose operational behaviour remains understandable even under degraded conditions.

"The objective of the app is not merely to display radar imagery. The objective is to expose radar acquisition behaviour clearly enough that downstream systems and operators can reason about radar freshness, availability, and operational state with confidence."

6. Overview

The ***App_Radar*** does not attempt to cover every radar protocol, transport variation, or vendor-specific radar behaviour that exists across the marine industry.

Instead, the app focuses on the radar acquisition, spoke handling, sweep generation, transport behaviour, and operational characteristics that matter most in real marine environments.

The objective is not protocol completeness for its own sake.

The objective is reliable radar acquisition, operational observability, deterministic replay capability, and predictable radar-service behaviour under both nominal and degraded operating conditions.

For this reason, the app prioritizes practical operational radar behaviours such as "***Radar Operational Variables***" monitoring.

The system is therefore designed around operational radar behaviour rather than exhaustive protocol coverage.

(a) Radar Operational Universe

Marine radar systems do not expose a single universal operational standard equivalent to GNSS NMEA sentence families.

Instead, radar systems typically expose a combination of:

- radar spokes,
- sweep timing behaviour,
- image buffers,
- radar state transitions,
- transport streams,
- control interfaces,
- and vendor-specific operational messages.

Collectively, these are referred to as the "***Radar State Variables***."

Different radar manufacturers, SDKs, transport layers, and hardware generations may expose these behaviours differently.

However, despite protocol differences, most marine radar systems ultimately revolve around the same operational concepts:

- radar acquisition,
- sweep progression,
- spoke delivery,
- image generation,
- radar mode control,
- transport timing,
- and operational liveness.

Collectively, these are referred to as the "***Radar Operational Variables***."

For this reason, the app focuses primarily on operational radar behaviour rather than exhaustive protocol abstraction.

The objective is not to normalize every vendor-specific radar implementation into a single artificial model.

The objective is to expose radar acquisition and radar operational behaviour consistently enough that downstream systems can reason about radar freshness, timing, and operational state predictably across different deployment environments.

(b) Parts of the Radar Module



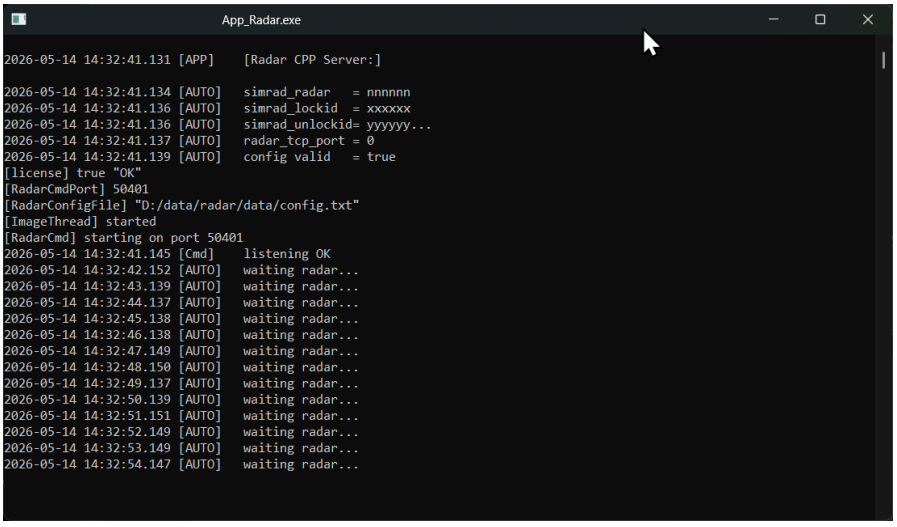
App_Radar is a fault-tolerant radar acquisition and radar-service module for marine radar systems. It interfaces with supported radar hardware to acquire, process, transport, replay, and expose radar operational data, including radar spokes, sweep behaviour, radar state, and generated radar imagery.

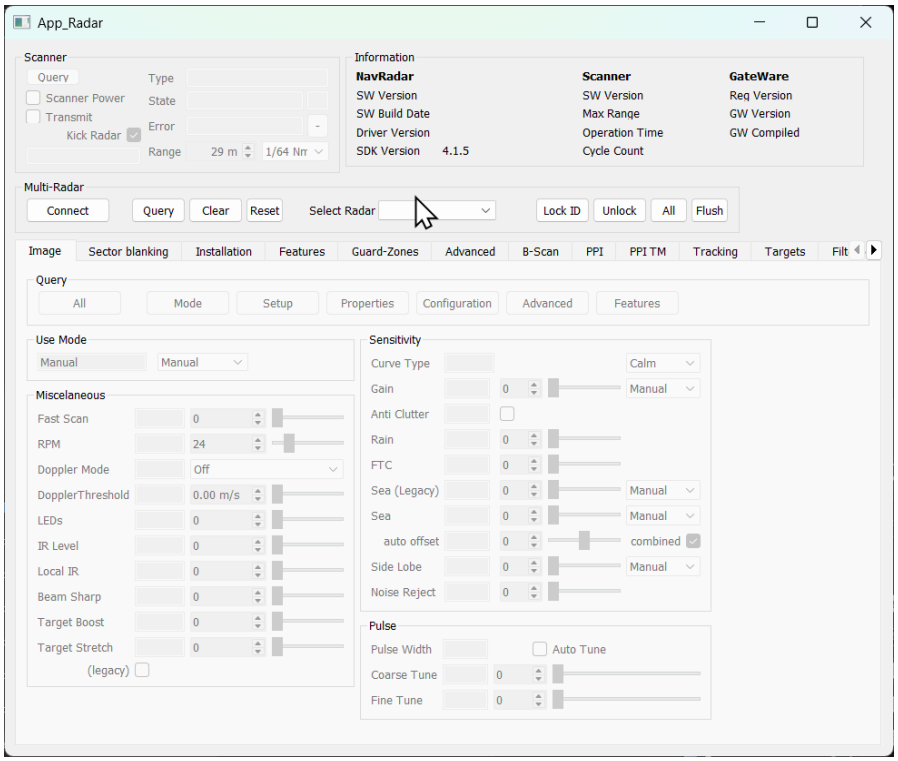
In simulation and replay modes, the app supports controlled spoke generation, deterministic radar replay, and repeatable operational testing under both nominal and degraded conditions.

Rather than treating radar imagery as automatically trustworthy, the app continuously exposes operational radar behaviour such as acquisition continuity, timing behaviour, transport state, image freshness, and radar liveness to downstream systems and operators.

By making radar acquisition behaviour observable rather than implicit, the app supports more reliable radar integration, operational troubleshooting, autonomy development, and repeatable radar-system testing across both live and simulated environments.

(1) Presentation of the App





The app presents radar operational behaviour through two coordinated interfaces:

- a real-time operational console, and
- a live radar control and visualization interface.

The operational console continuously exposes internal radar-service activity such as:

- radar acquisition state,
- connection behaviour,
- replay activity,
- transport status,
- radar mode transitions,
- and fault-tolerant runtime behaviour.

Collectively, these are referred to as the "***Radar Operational State***."

This interface is intended primarily for operational observability, diagnostics, replay analysis, and integration troubleshooting.

The radar control and visualization interface exposes live radar-system behaviour and operational state through a set of real-time operational pills and radar-state indicators.

The top-line pills primarily expose operational behaviour, including:

- radar mode,
- acquisition activity,
- connection state,
- data freshness,
- and runtime liveness.

Collectively, these are referred to as the "***Radar Operational State***."

The second-line pills expose evaluated operational state derived from live radar acquisition behaviour, timing consistency, freshness analysis, transport behaviour, and radar-state coherence.

These indicators are intentionally operational rather than cosmetic.

The objective is not merely to show that radar imagery exists.

The objective is to make radar acquisition behaviour observable to both operators and downstream systems so that degraded, stale, or inconsistent radar operation can be identified explicitly rather than inferred indirectly from the displayed radar image alone.

(2) Real-time Operational Console

The app includes a real-time operational console intended for radar acquisition monitoring, diagnostics, replay observation, and runtime troubleshooting.

Unlike traditional radar displays that focus primarily on rendered imagery, the operational console exposes the underlying radar-service behaviour directly as the system runs.

This includes live visibility into "***Radar Operational State***."

The console is intentionally simple, text-oriented, and continuously updated in real time.

Its purpose is not cosmetic presentation.

Its purpose is to make radar-service behaviour operationally observable during both nominal and degraded operating conditions.

This becomes particularly important during:

- acquisition delays,
- connection interruptions,
- stale radar behaviour,
- replay analysis,
- and transient runtime failures that may not be immediately visible through radar imagery alone.

By exposing runtime radar behaviour directly, the console allows integrators and operators to reason about radar acquisition state, service liveness, and operational progression without relying solely on the rendered radar image.

(3) Live Radar Control and Visualization Interface

The app includes a real-time radar control, radar-state visualization, and live operational monitoring during acquisition and replay.

(c) Core Responsibilities

The **radar-head module** is the system's radar acquisition and radar-service layer:

- It interfaces directly with supported radar hardware and radar transport streams,
- acquires and processes radar spokes, sweep behaviour, radar state, and generated radar imagery,
- monitors acquisition continuity, timing behaviour, transport integrity, and radar operational state,
- detects degraded operational conditions such as stale sweeps, spoke interruptions, delayed imagery, transport instability, and acquisition stalls,
- and exposes operational radar behaviour transparently to downstream systems, replay tools, and runtime monitoring interfaces.

The module is designed to behave consistently across REAL, SIM, and replay operating modes while maintaining operational visibility into radar acquisition behaviour throughout the radar pipeline.

Its responsibility is not to interpret radar targets or perform navigation decisions.

Its responsibility is to acquire, monitor, replay, and expose radar operational behaviour predictably, transparently, and fault-tolerantly to the rest of the radar and navigation stack.

(d) Reliability by Design

- Operates as an independent radar-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 radar-head module encounters operational failure, it can be restarted independently **without** requiring restart of unrelated systems such as navigation, AIS, engine, steering, autonomy, or user-interface layers.

(e) SIM vs REAL Operation

(1) SIM Mode (not yet fully implemented and exposed)

- No physical radar hardware required,
- supports deterministic and repeatable radar behaviour,
- simulation and replay scenarios provide operational conditions that help anticipate and reproduce REAL-mode acquisition and transport behaviours,
- and is well suited for integration testing, replay analysis, demonstrations, and radar-service validation.

(2) REAL Mode

- Interfaces directly with supported radar hardware and live radar transport streams,
- uses the same operational radar-service pathways as SIM and replay modes to minimize behavioural divergence,
- and maintains a consistent radar-service abstraction layer so that downstream systems remain operationally unchanged across REAL, SIM, and replay operation.

(f) Why This Matters

By isolating radar acquisition and radar operational behaviour inside the radar-head module:

- downstream systems avoid direct dependence on vendor-specific radar behaviour,
- radar acquisition, replay, and transport handling remain centralized and testable,
- radar hardware or SDK changes are less likely to ripple unpredictably across the wider system,
- and replay analysis, troubleshooting, and operational testing become significantly more repeatable and manageable.

This separation helps keep downstream radar consumers simpler, cleaner, and operationally more predictable.

(g) Intended Use

The app is designed to operate alongside other white-label modules such as:

- navigation and autopilot systems,
- AIS services,
- operator interfaces,
- camera and sensor systems,
- boat-motion and simulation modules,
- and downstream radar visualization or replay tools.

Each module focuses on a specific operational responsibility, while the radar-head module focuses specifically on radar acquisition, radar operational behaviour, replay-ability, and radar-service observability.

7. The Real Mode

The REAL Mode represents the operational state in which the radar-head module interfaces directly with live radar hardware and live radar transport streams rather than internally generated simulation or replay data.

In this mode, the app operates as a deterministic radar acquisition and radar-service layer for the surrounding system. Radar spokes, sweep behaviour, radar state transitions, and generated radar imagery originate from connected radar hardware and live acquisition pipelines.

The module does not attempt to reinterpret radar targets or perform downstream navigation decisions.

Instead, it focuses on acquiring, monitoring, replaying, transporting, and exposing operational radar behaviour consistently and predictably to downstream systems and operators.

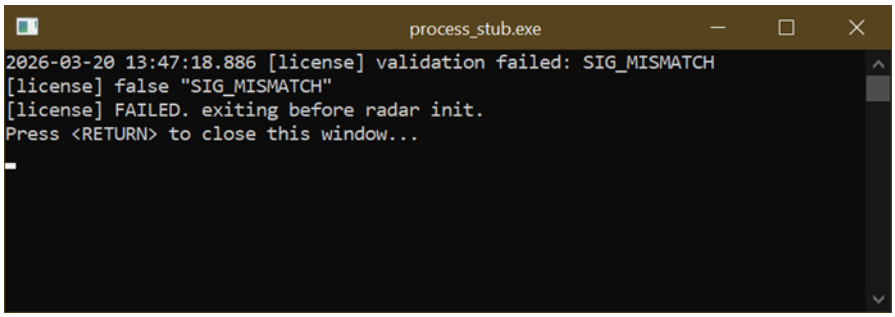
REAL Mode therefore uses the same radar-service architecture and operational pipeline used in SIM and replay modes, with live radar acquisition replacing internally generated radar behaviour.

The design philosophy remains unchanged:

The radar source may change, but the operational handling, timing behaviour, replay-ability, and radar-service observability remain consistent across operating modes.

(a) Typical Real Mode Workflow

(1) License Validation



Before radar acquisition begins, the app performs license validation during startup initialization.

This validation occurs before:

- radar hardware initialization,
- radar transport setup,
- spoke acquisition,
- or radar-service activation.

If validation succeeds:

- the radar-service initialization sequence continues normally,
- radar acquisition threads are started,
- and REAL Mode operation proceeds.

If validation fails:

- radar initialization is aborted safely,
- no live radar acquisition is started,
- and the operational console exposes the validation failure state explicitly.

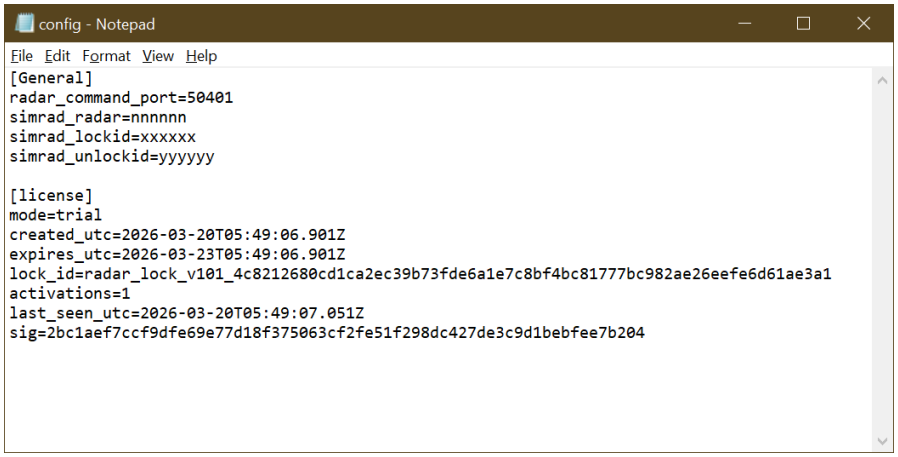
Typical failure reasons may include:

- signature mismatch,
- invalid license state,
- missing runtime authorization,
- or incompatible license configuration.

The app intentionally performs validation early in the startup sequence so that invalid operational states are detected before radar-service activation occurs.

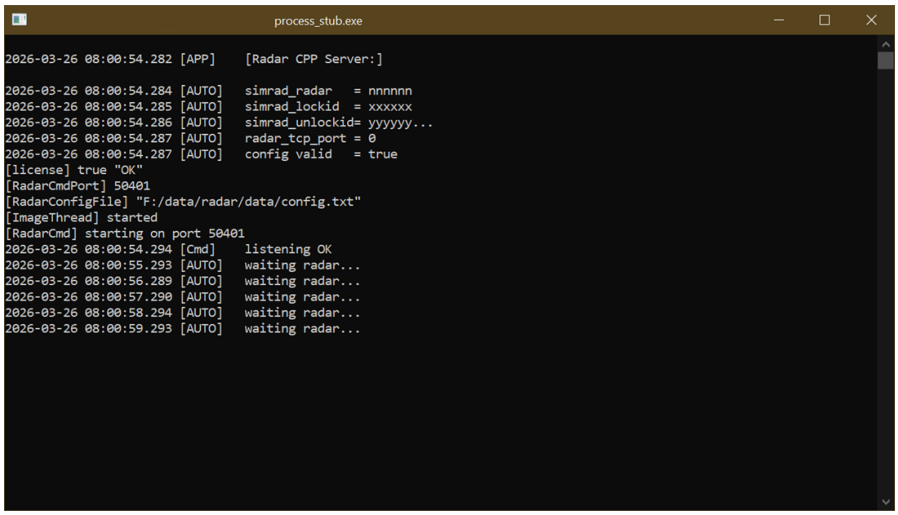
If the app is launched for the first time, it will automatically generate a temporary evaluation license inside the [`config.txt`].

This temporary license is valid for 3 days and allows the radar-head module to operate normally during evaluation, integration, and testing.

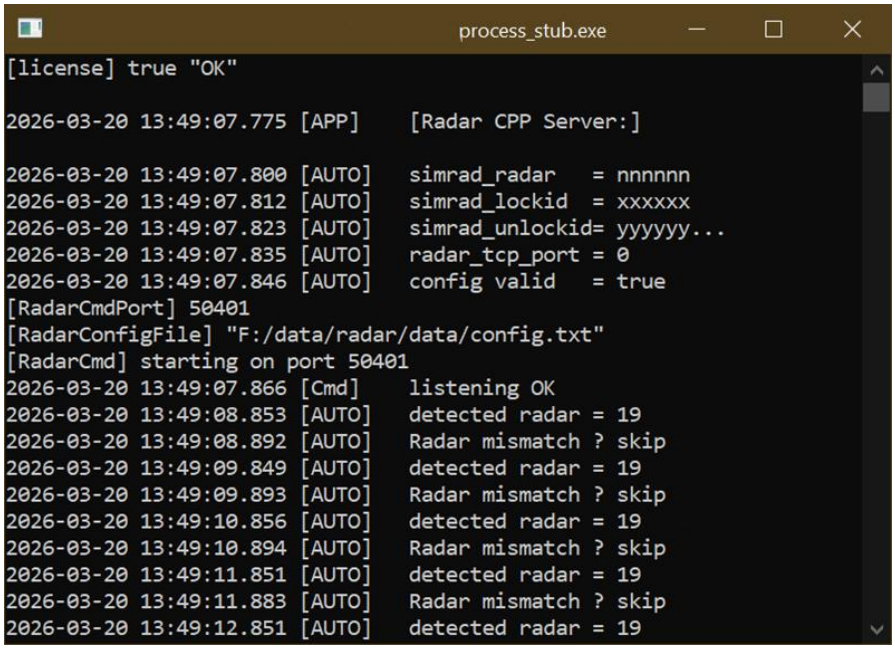


After the evaluation period expires, license validation will fail and REAL Mode radar acquisition will no longer start.

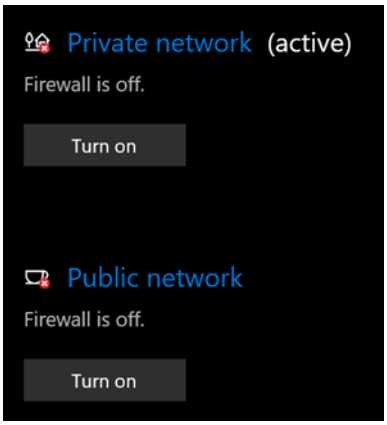
To generate a fresh temporary evaluation license, simply delete the existing [*`config.txt`*] and restart the app. A new temporary evaluation license will then be created automatically during startup initialization.



(2) Initial Startup



After successful license validation, the app begins the radar-service startup sequence. If you encounter too much in-activities, disabling the firewall temporarily is a possible get-around [do re-enable the firewalls after successful tests].



During initialization, the operational console exposes live startup behaviour such as:

- radar configuration loading,
- radar command-port initialization,
- radar-service thread startup,
- runtime parameter validation,
- and early radar detection activity.

Once initialization completes successfully:

- the radar command interface begins listening,
- the radar acquisition pipeline becomes active,
- and the app starts monitoring for compatible live radar hardware and transport activity.



The startup sequence is intentionally exposed through the operational console so that initialization behaviour, radar detection progress, and early operational issues remain directly observable during deployment and troubleshooting.

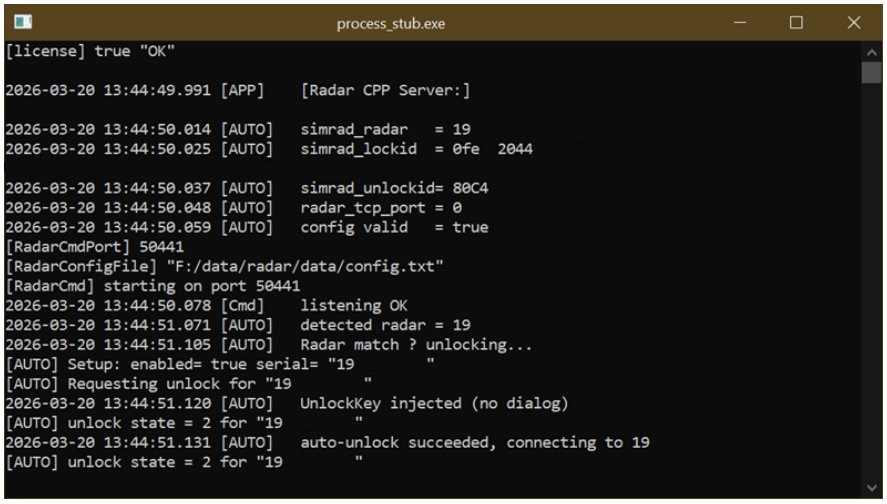
(3) Setting & Reading Setup in [config.txt]



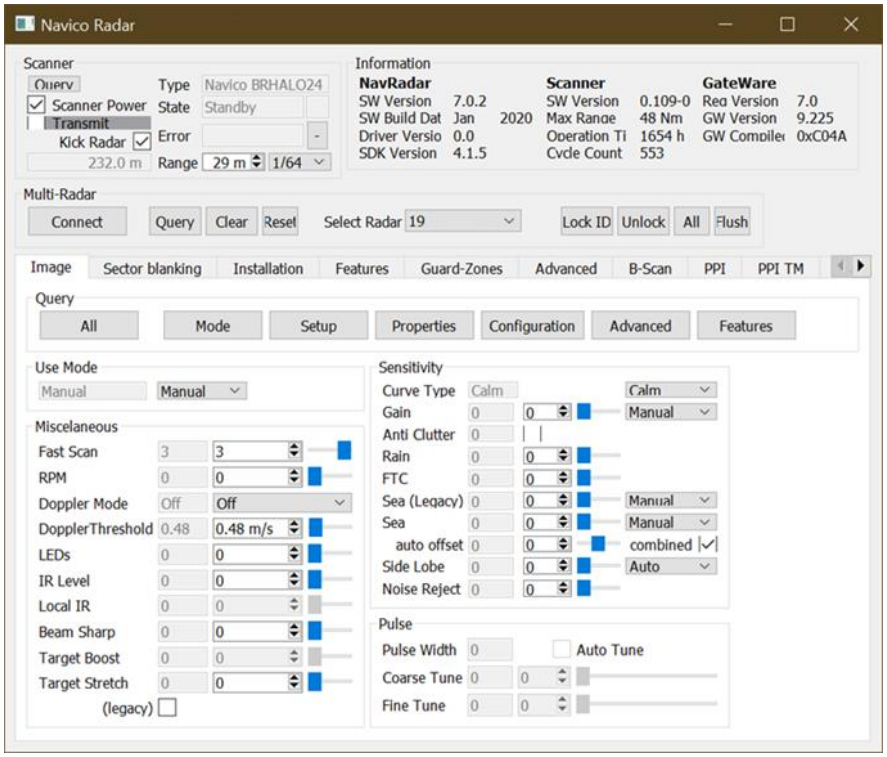
Locate the [config.txt] and update the following:

- radar tcp command port,
- radar antenna id,
- radar lockid,
- radar unlockid,

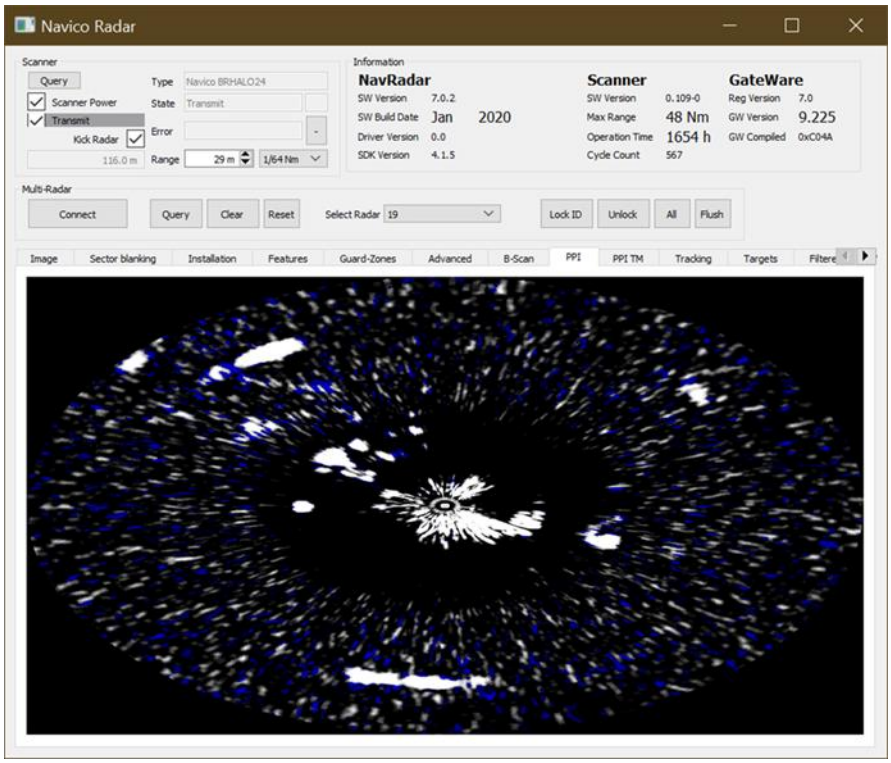
Save the file and restart the app.

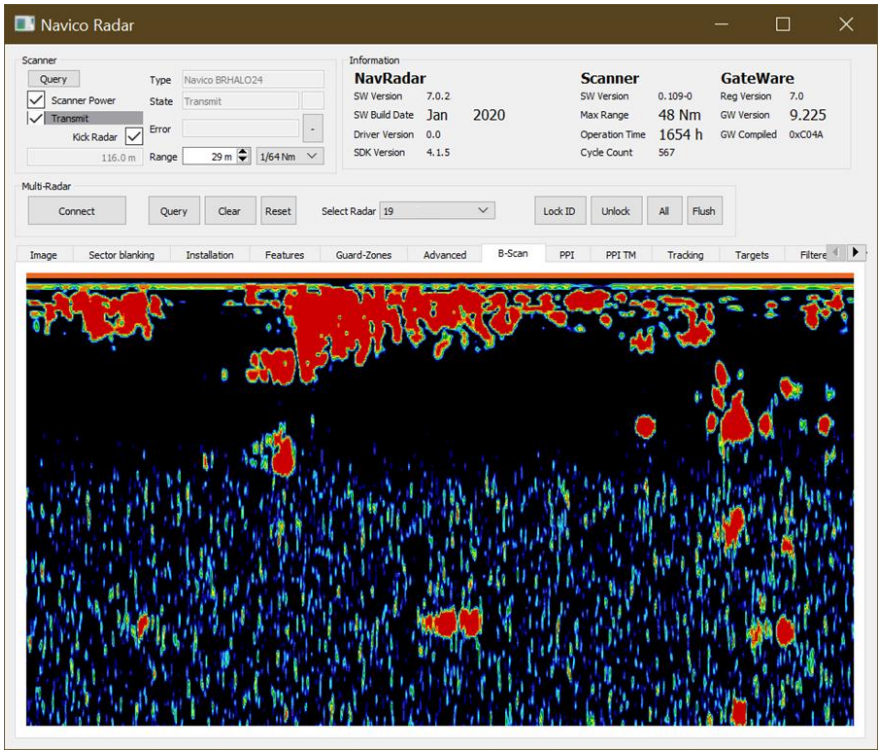


The radar antenna would be unlocked for reading of spoke data.

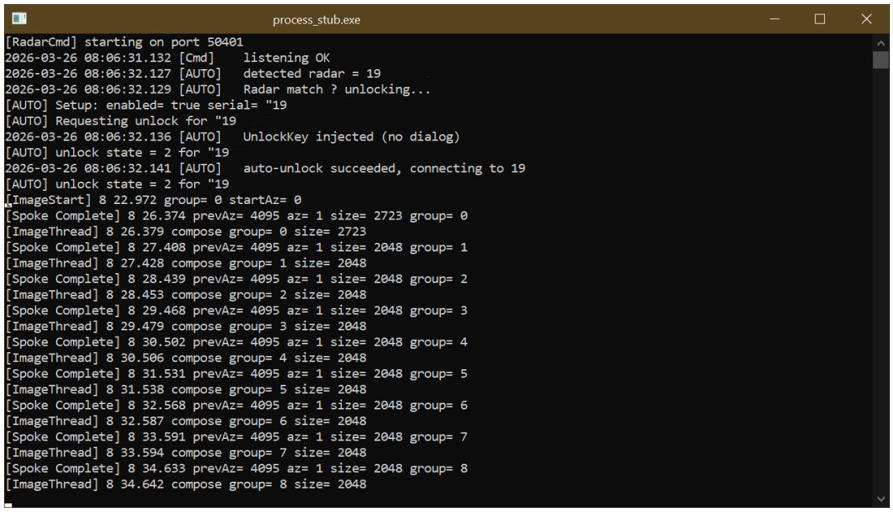


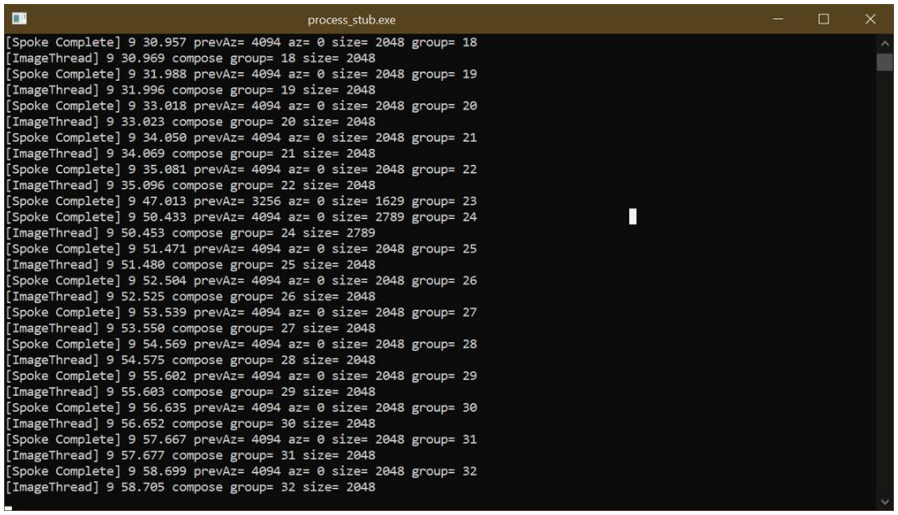
Try transmitting.





(4) Spoke Data Collection and Image Generation





When the radar antenna enters transmit mode, the app begins acquiring live radar spoke data from the radar acquisition stream.

Each incoming spoke contains radar echo information representing reflected radar energy observed during the current sweep position.

The app continuously:

- collects incoming spokes,
- evaluates spoke timing and acquisition continuity,
- processes echo-strength information,
- and consolidates spoke data into progressive sweep construction buffers.

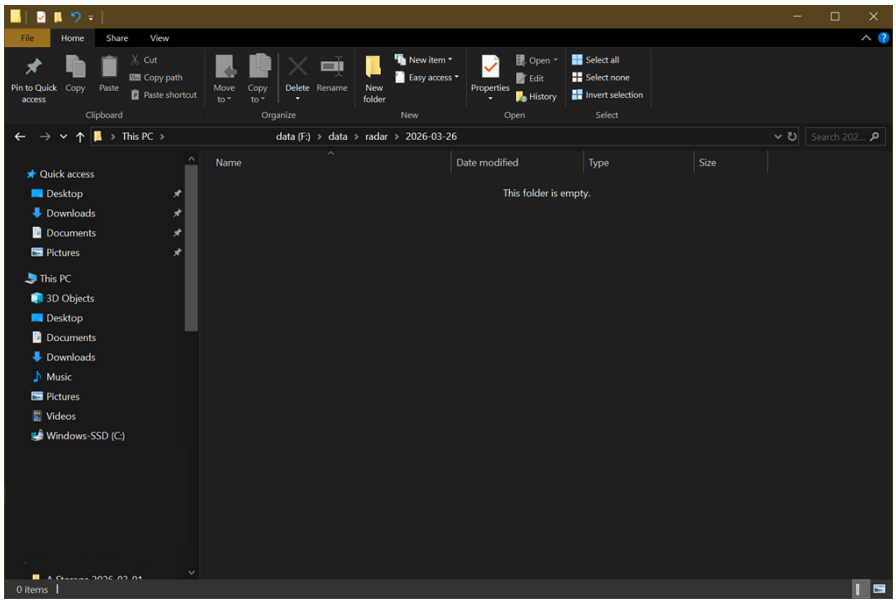
Once a complete radar sweep cycle has been acquired, the app generates a radar image corresponding to the collected spoke data for that sweep interval.

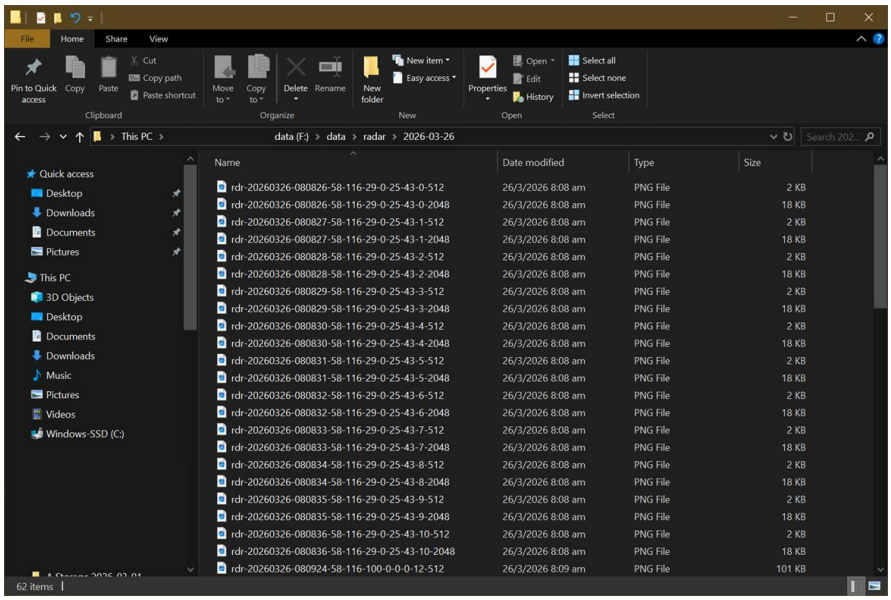
The generated radar images are then deposited into the:

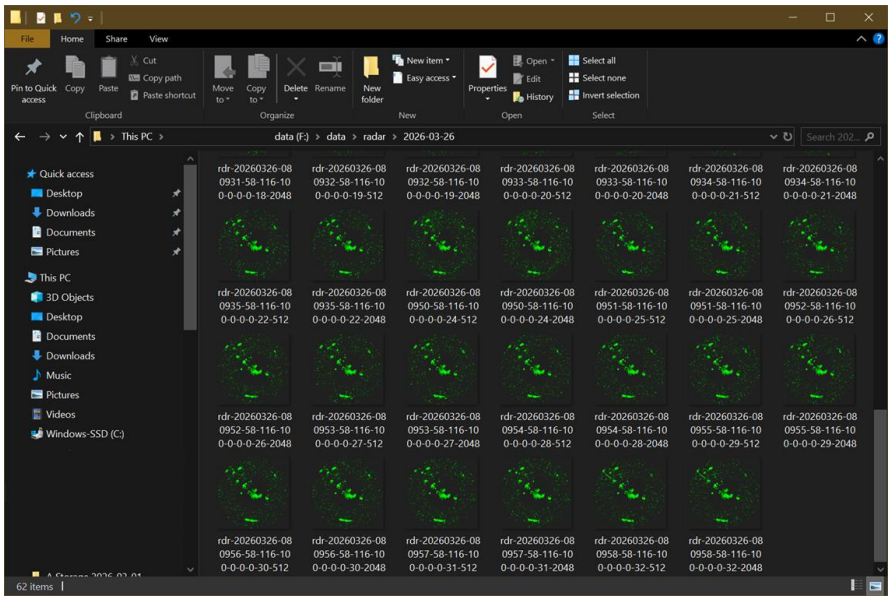
[*data/radar/{date}/*]

directory using a timestamped filename structure such as:

[*rdr, yyyymmdd, hhmmss, rpm, range_m, gain, ftc, sea, rain, scan_no, img_size_px, .png*]


File
Home
Share
View
Pin to Quick access
Copy
Paste
Cut
Copy path
Paste shortcut
Clipboard
Move to
Copy to
Delete
Rename
Organize
New item
Easy access
New folder
New
Properties
Open
Edit
History
Open
Select all
Select none
Invert selection
Select
This PC
data (F:)
data
radar
2026-03-26
Search 202...
Quick access
Desktop
Downloads
Documents
Pictures
This PC
3D Objects
Desktop
Documents
Downloads
Music
Pictures
Videos
Windows-SSD (C:)
Name
Date modified
Type
Size
This folder is empty.
0 items


File
Home
Share
View
Pin to Quick access
Copy
Paste
Cut
Copy path
Paste shortcut
Clipboard
Move to
Copy to
Delete
Rename
Organize
New item
Easy access
New folder
New
Properties
Open
Edit
History
Open
Select all
Select none
Invert selection
Select
This PC
data (F:)
data
radar
2026-03-26
Search 202...
Quick access
Desktop
Downloads
Documents
Pictures
This PC
3D Objects
Desktop
Documents
Downloads
Music
Pictures
Videos
Windows-SSD (C:)
Name
Date modified
Type
Size
rdr-20260326-080826-58-116-29-0-25-43-0-512 26/3/2026 8:08 am PNG File 2 KB
rdr-20260326-080826-58-116-29-0-25-43-0-2048 26/3/2026 8:08 am PNG File 18 KB
rdr-20260326-080827-58-116-29-0-25-43-1-512 26/3/2026 8:08 am PNG File 2 KB
rdr-20260326-080827-58-116-29-0-25-43-1-2048 26/3/2026 8:08 am PNG File 18 KB
rdr-20260326-080828-58-116-29-0-25-43-2-512 26/3/2026 8:08 am PNG File 2 KB
rdr-20260326-080828-58-116-29-0-25-43-2-2048 26/3/2026 8:08 am PNG File 18 KB
rdr-20260326-080829-58-116-29-0-25-43-3-512 26/3/2026 8:08 am PNG File 2 KB
rdr-20260326-080829-58-116-29-0-25-43-3-2048 26/3/2026 8:08 am PNG File 18 KB
rdr-20260326-080830-58-116-29-0-25-43-4-512 26/3/2026 8:08 am PNG File 2 KB
rdr-20260326-080830-58-116-29-0-25-43-4-2048 26/3/2026 8:08 am PNG File 18 KB
rdr-20260326-080831-58-116-29-0-25-43-5-512 26/3/2026 8:08 am PNG File 2 KB
rdr-20260326-080831-58-116-29-0-25-43-5-2048 26/3/2026 8:08 am PNG File 18 KB
rdr-20260326-080832-58-116-29-0-25-43-6-512 26/3/2026 8:08 am PNG File 2 KB
rdr-20260326-080832-58-116-29-0-25-43-6-2048 26/3/2026 8:08 am PNG File 18 KB
rdr-20260326-080833-58-116-29-0-25-43-7-512 26/3/2026 8:08 am PNG File 2 KB
rdr-20260326-080833-58-116-29-0-25-43-7-2048 26/3/2026 8:08 am PNG File 18 KB
rdr-20260326-080834-58-116-29-0-25-43-8-512 26/3/2026 8:08 am PNG File 2 KB
rdr-20260326-080834-58-116-29-0-25-43-8-2048 26/3/2026 8:08 am PNG File 18 KB
rdr-20260326-080835-58-116-29-0-25-43-9-512 26/3/2026 8:08 am PNG File 2 KB
rdr-20260326-080835-58-116-29-0-25-43-9-2048 26/3/2026 8:08 am PNG File 18 KB
rdr-20260326-080836-58-116-29-0-25-43-10-512 26/3/2026 8:08 am PNG File 2 KB
rdr-20260326-080836-58-116-29-0-25-43-10-2048 26/3/2026 8:08 am PNG File 18 KB
rdr-20260326-080924-58-116-100-0-0-0-12-512 26/3/2026 8:09 am PNG File 101 KB
62 items



This allows radar acquisition behaviour, sweep progression, and generated imagery to remain replay-able, reviewable, and operationally traceable across time.

(b) Summary

REAL Mode represents live operation of the radar-head module using physical radar hardware and live radar acquisition streams.

The radar-service architecture, acquisition pipeline, replay handling, operational monitoring, and runtime behaviour remain consistent with those used in SIM and replay modes. Only the source of the radar acquisition data changes.

By maintaining consistent operational handling across REAL, SIM, and replay operation, the module allows behaviours observed during simulation and replay analysis to translate directly into live radar operating conditions.

Simulation and replay support repeatable observation.

REAL Mode exposes live radar behaviour.

8. Price

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

The pricing of this white-label module 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

- **Radar Acquisition Service**

  Interfaces directly with supported radar hardware and radar transport streams.

  Receives live radar spokes, sweep updates, radar-state information, and operational runtime behaviour from REAL, SIM, or replay sources.

  Buffers incoming acquisition data before forwarding it into the radar processing pipeline.

- **Radar Processing & Image Generation Engine**

  Processes incoming radar spokes and sweep behaviour.

  Handles:

  - spoke consolidation
  - sweep progression
  - echo-strength evaluation
  - image construction
  - and radar image generation

  Generates radar imagery corresponding to the current radar acquisition cycle.

- **Radar Operational State Engine**

  Maintains consolidated radar operational state information including:

  - radar mode
  - transmit state
  - range settings
  - sweep activity
  - acquisition continuity
  - image freshness
  - and radar runtime status

  Maintains observable operational state across REAL, SIM, and replay operation.

- **Radar Timing & Acquisition Monitor**

  Continuously monitors radar acquisition behaviour across time.

  Detects operational conditions such as:

  - stale sweep behaviour
  - spoke interruptions
  - delayed image updates
  - transport jitter
  - acquisition stalls
  - and timing drift across the radar pipeline

  Feeds operational monitoring information into runtime indicators, replay logs, and operational diagnostics.

- **Radar Hardware Abstraction Layer (HAL)**

  Provides a unified radar acquisition interface independent of radar vendor, SDK, or transport implementation.

  Supports:

  - live radar hardware
  - replay acquisition streams
  - and internally generated simulation behaviour

  Allows REAL, SIM, and replay modes to operate through a consistent radar-service architecture.

- **Simulation & Replay Engine**

  Supports deterministic radar replay and controlled radar simulation behaviour.

  Supports:

  - recorded spoke replay
  - controlled sweep generation
  - acquisition interruption scenarios
  - timing degradation scenarios
  - and transport-behaviour simulation

  Allows repeatable radar-system testing and replay analysis under controlled operating conditions.

- **Output Distribution Service**

  Publishes radar operational behaviour and generated radar imagery to:

  - runtime monitoring interfaces
  - downstream radar consumers
  - replay systems
  - and radar visualization interfaces

Provides a centralized radar-service layer for downstream operational use.

- **Recording & Replay Service**

  Supports recording of radar operational behaviour and generated radar imagery.

  Provides:

  - timestamp tagging
  - file rotation
  - replay indexing
  - and deterministic replay capability for troubleshooting and operational analysis

  Allows transient radar acquisition behaviour to remain reviewable and reproducible across time.

- **Runtime Heartbeat & Fault-Tolerance Service**

  Provides operational liveness monitoring for the radar-service runtime environment.

  Supports:

  - heartbeat monitoring
  - stale acquisition detection
  - runtime pause handling
  - restart/kill markers
  - and automatic restart behaviour under degraded runtime conditions

  Allows radar-service behaviour to remain externally observable during both nominal and degraded operation.

- **Configuration & Runtime Settings Manager**

  Loads radar operational configuration parameters including:

  - radar transport settings
  - radar unlock configuration
  - runtime operational settings
  - recording parameters
  - replay behaviour
  - and simulation configuration

  Allows deployment-specific tuning without modifying the core radar-service architecture.

(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_Radar_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_Radar.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/radar/data**] directory for the correct TCP Cmd Port [at the beginning, do keep it as default] and do a restart of the App.

(c) Installation & Directories

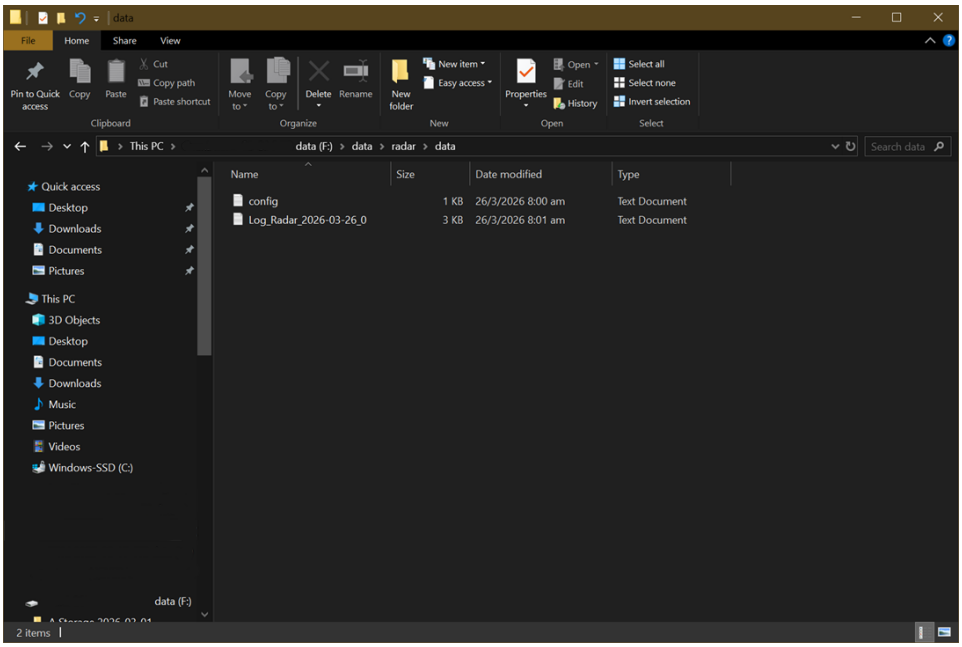
- Application root, e.g.,
  <drive:>\Apps\App_Radar_Group\App_Radar
- Data directory: *<drive:>\data\radar\{date}* (auto-created)
- Log directory: *<drive:>\data\radar\data*
- Config file: *<drive:>\data\radar\data\config.txt*
- Splash Logo file:
  <drive:>\data\radar\data\app_radar_logo.png
- App Radar Icon file: *<drive:>\data\radar\data\app_radar.ico*

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

It has a [***data\radar\{date}***] dir and [*data\radar**<u>***data***</u>], 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**.



⚠ If it fails on another PC

Most common issue:

Missing Microsoft runtime

Error like:

```
VCRUNTIME140.dll missing
MSVCP140.dll missing
```

Fix (very standard)

Install:

```
Microsoft Visual C++ Redistributable 2015-2022 (x64)
```

That's it.

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 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 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*

Inside the [***data***] directory, the app maintains a [***config.txt***] file containing operational radar-service configuration parameters.

Most runtime behaviour, radar-service settings, replay behaviour, and operational options can be adjusted directly through this configuration file without modifying the application source code.

A typical configuration may contain sections such as:

```
[General]

radar_command_port=50401

simrad_radar=nnnnnn

simrad_lockid=xxxxxx

simrad_unlockid=yyyyyy

simrad_file_format=rdr,yyyymmdd,hhmmss,rpm,range_m,gain,ftc,sea,rain,scan_no,img_size_px,.
png

simrad_reduced_filesize=512

simrad_title=App_Radar

[license]

…
```

The primary configuration parameters are described below.

(a) General

(1) radar_command_port

Defines the TCP command port used by the radar-service runtime. Default is **50401**.

This command interface allows downstream tools, replay systems, monitoring interfaces, and operational utilities to communicate with the radar-head module during runtime.

(2) simrad_radar

Defines the expected radar identifier or radar target configuration used during radar detection and initialization.

This setting helps the app determine which radar hardware instance should be acquired during startup.

(3) simrad_lockid

Defines the radar lock identifier associated with radar initialization and controlled radar acquisition behaviour.

Used during radar matching and operational startup validation.

(4) simrad_unlockid

Defines the radar unlock identifier used during controlled radar unlock and acquisition initialization.

Used during automatic radar acquisition startup sequences.

(5) simrad_file_format

Defines the filename structure used when generated radar imagery is deposited into the radar recording directory.

The filename may include runtime radar metadata such as:

- date,
- time,
- radar RPM,
- range,
- gain settings,
- FTC settings,
- sea clutter settings,
- rain clutter settings,
- scan number,
- and generated image size.

This structure allows recorded radar imagery to remain operationally traceable and replay-friendly across time.

(6) simrad_reduced_filesize

Defines the reduced radar image size used for lower-resolution image generation and storage. The original size is **2048** pixels.

Useful for:

- reduced storage usage,
- faster replay handling,
- lighter-weight operational review,
- and reduced transport overhead during downstream processing.

(7) simrad_title

Defines the runtime title displayed by the radar-head application and associated runtime interfaces.

Useful for deployment identification and multi-instance operational environments.

(b) License

Contains runtime licensing and operational authorization information used during startup validation and REAL Mode initialization.

The license section is validated during startup before radar acquisition begins.

13. Command Port Protocol

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

(a) Core GET/CMD/SET commands

(1) `get health`

(2) `get radar.state`

(3) `set radar.range <nnn>`

(4) `set radar.gain <nn>`

(5) `set radar.rain <nn>`

(6) `set radar.sea <nn>`

(7) `set radar.ftc <nn>`

(8) `set radar restart`

(9) `set radar kill`

(10) `set radar.power <kick/unkick>`

(11) `set radar.power off`

(12) `set radar.power standby`

(13) `set radar.power transmit`

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

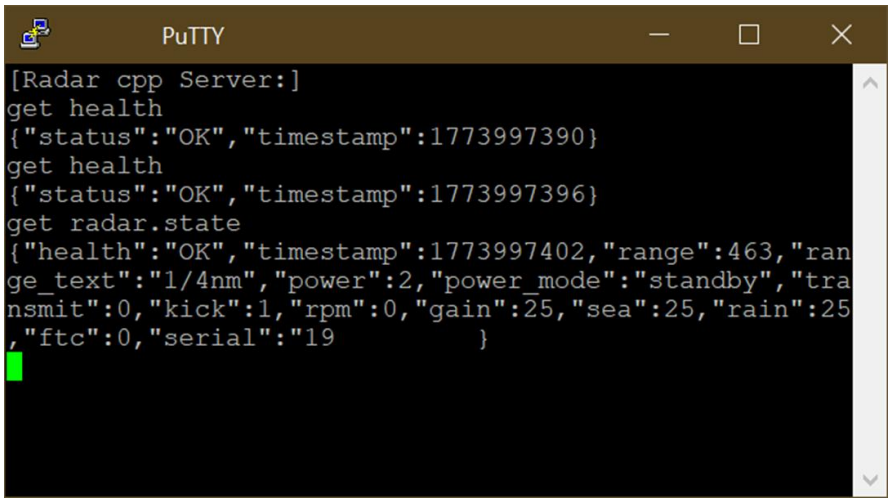
Using a port reader such as ***PuTTY*** app (copyrighted by *Simon Tatham*). Key-in the following detail:

- Host Name: localhost or **`127.0.0.1`**
- Port: **`50401`** (or your specified port address in the [***`config.txt`***])
- Connection Type: Other (*Raw*)

Click [*Open*].



(1) *get health* [replies in JSON]



[Radar cpp Server:]

get health
{"status":"OK","timestamp":1773997390}

get health
{"status":"OK","timestamp":1773997396}

get radar.state
{"health":"OK","timestamp":1773997402,"range":463,
"range_text":"1/4nm","power":2,"power_mode":"standby",
"transmit":0,"kick":1,"rpm":0,"gain":25,"sea":25,
"rain":25,"ftc":0,"serial":"19"}

Returns a lightweight runtime health response from the radar-head service.

Typical usage:

get health

Typical JSON reply:

{"status":"OK","timestamp":1773997390}

The command is intended primarily for:

- runtime liveness checks,
- service heartbeat monitoring,
- external watchdog supervision,
- and lightweight operational polling.

Unlike:

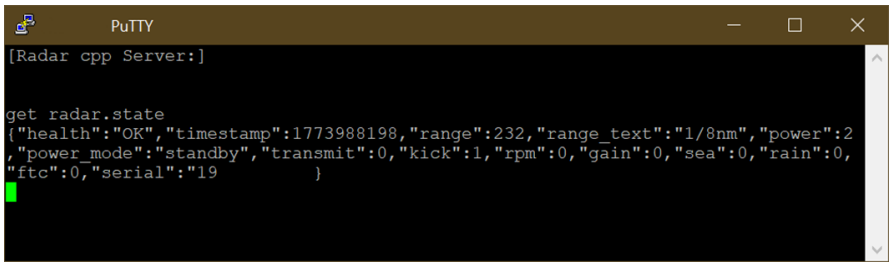
get radar.state

which returns detailed radar operational information, the:

get health

command provides only minimal runtime health information suitable for high-frequency monitoring and external fault-tolerant supervision systems.

(2) *get radar.state* [replies in JSON]



[Radar cpp Server:]

get radar.state
{"health":"OK","timestamp":1773988198,"range":232,"range_text":"1/8nm","power":2,
"power_mode":"standby","transmit":0,"kick":1,"rpm":0,"gain":0,"sea":0,"rain":0,
"ftc":0,"serial":"19"}

Returns the current operational radar state as a JSON response.

Typical information returned may include:

- radar health state,
- runtime timestamp,
- current radar range,
- range text representation,
- power state,
- transmit state,
- kick/startup state,
- radar RPM,
- gain settings,
- sea clutter settings,
- rain clutter settings,
- FTC settings,
- and detected radar serial information.

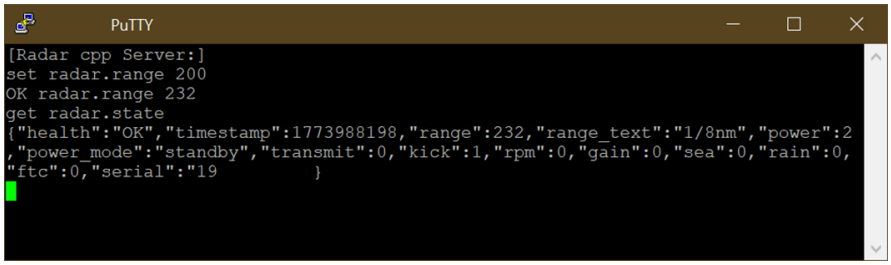
This command is primarily used for:

- runtime monitoring,
- operational diagnostics,
- replay verification,
- and downstream radar-service integration.

The response is intentionally lightweight and machine-readable so that downstream systems can query live radar operational behaviour consistently during runtime.

(3) *set radar.range <nnn>* [replies in Text]

Example I:



[Radar cpp Server:]

set radar.range 200
OK radar.range 232

get radar.state
{"health":"OK","timestamp":1773988198,"range":232,"range_text":"1/8nm","power":2,
"power_mode":"standby","transmit":0,"kick":1,"rpm":0,"gain":0,"sea":0,"rain":0,
"ftc":0,"serial":"19"}

Sets the operational radar range used by the live radar acquisition system.

The command accepts a requested radar range value. The radar hardware or SDK may internally quantize the requested value to the nearest supported operational range.

Typical usage:

set radar.range 200

Typical text reply:

OK radar.range=232 (~1/8nm)

In this example:

- the requested range was 200 m,
- but the radar hardware internally selected 232 m,
- which corresponds approximately to a 1/8 nautical mile operational range.

Approximate operational range mappings (sample supported operational radar ranges) may include:

- *1/64nm ≈ 29m*
- *1/48nm ≈ 38m*
- *1/37nm ≈ 50m*
- *1/32nm ≈ 58m*
- *1/24nm ≈ 77m*
- *1/16nm ≈ 116m*
- *1/8nm ≈ 232m*
- *1/4nm ≈ 463m*
- *1/2nm ≈ 926m*
- *3/4nm ≈ 1389m*
- *1nm ≈ 1852m*
- *1.5nm ≈ 2778m*
- *2nm ≈ 3704m*
- *3nm ≈ 5556m*
- *4nm ≈ 7408m*
- *6nm ≈ 11112m*
- *8nm ≈ 14816m*
- *12nm ≈ 22224m*
- *16nm ≈ 29632m*
- *24nm ≈ 44448m*
- *36nm ≈ 66672m*
- *48nm ≈ 88896m*
- *96nm ≈ 177792m*

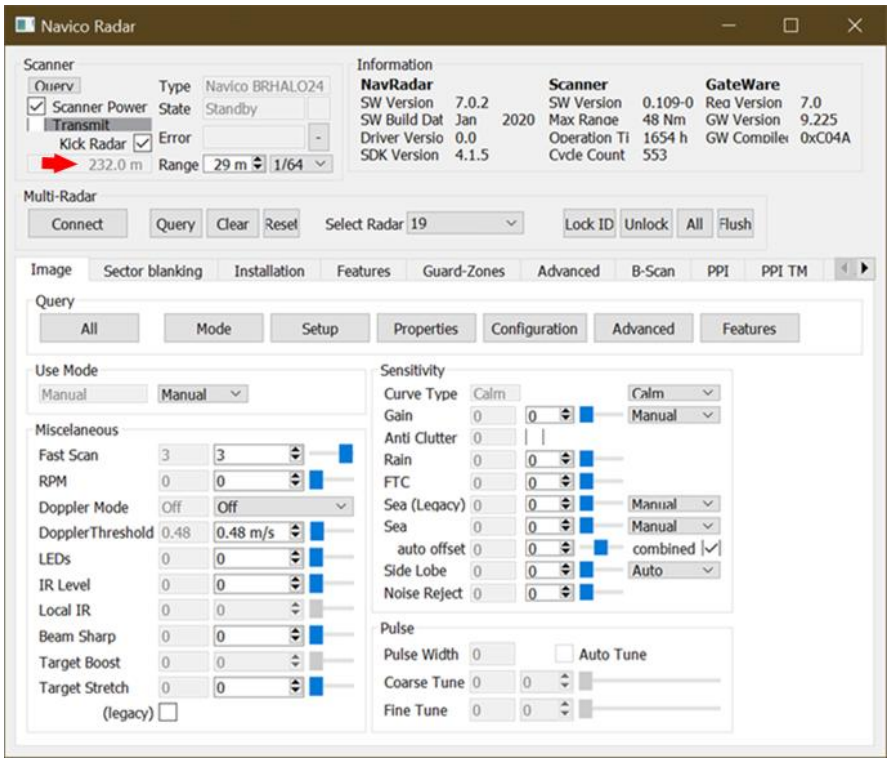
The radar hardware may internally quantize the requested value to the nearest supported operational radar range.

The resulting radar range affects:

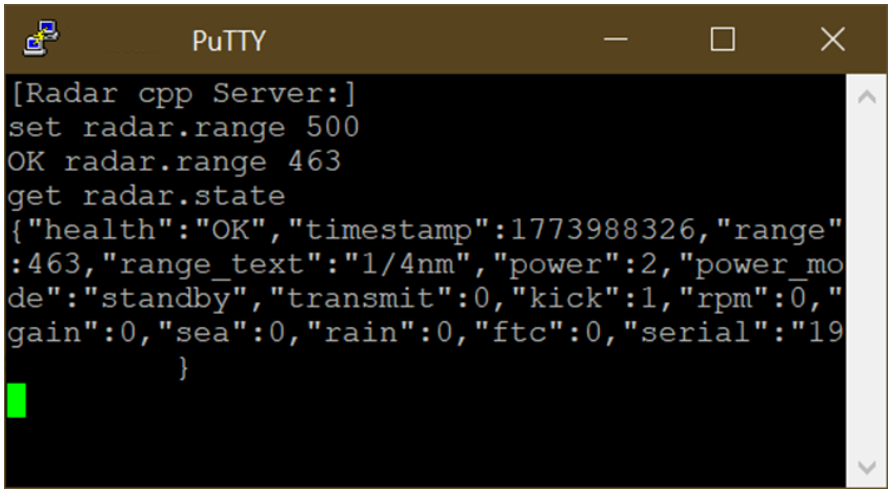
- radar sweep scaling,
- spoke acquisition coverage,
- generated radar imagery,
- and downstream radar visualization behaviour.

Range transitions are exposed through the radar operational state so that downstream systems and runtime interfaces remain synchronized with the active radar acquisition environment.

A check on its current [***randar.state***], returns the current operational radar state as a JSON response.



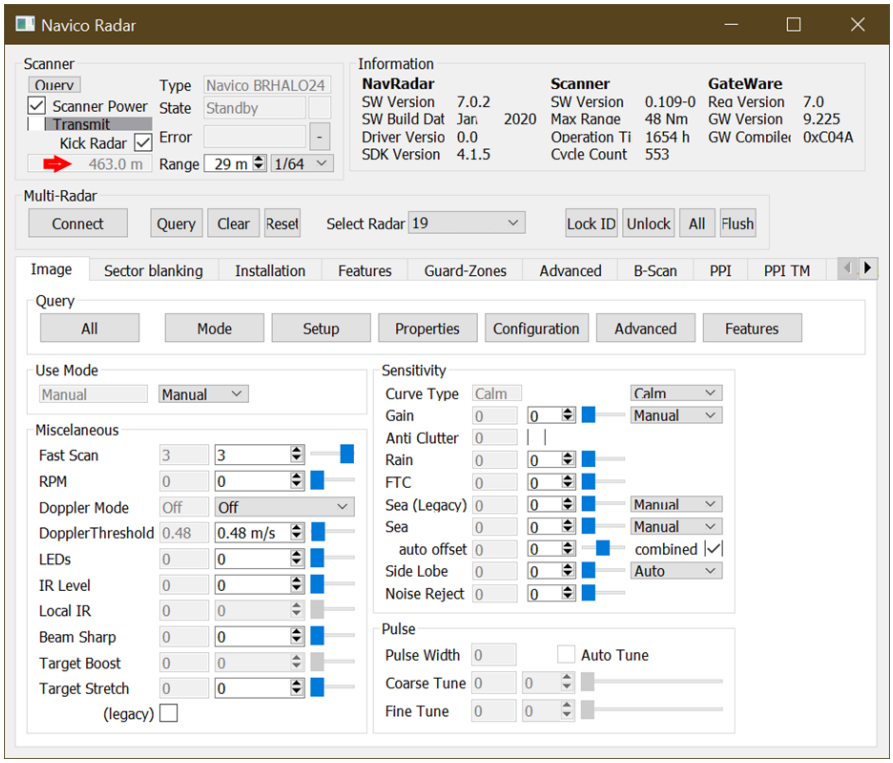
Example II:



[Radar cpp Server:]

set radar.range 500
OK radar.range 463

get radar.state
{"health":"OK","timestamp":1773988326,"range":463,"range_text":"1/4nm",
"power":2,"power_mode":"standby","transmit":0,"kick":1,"rpm":0,
"gain":0,"sea":0,"rain":0,"ftc":0,"serial":"19"}



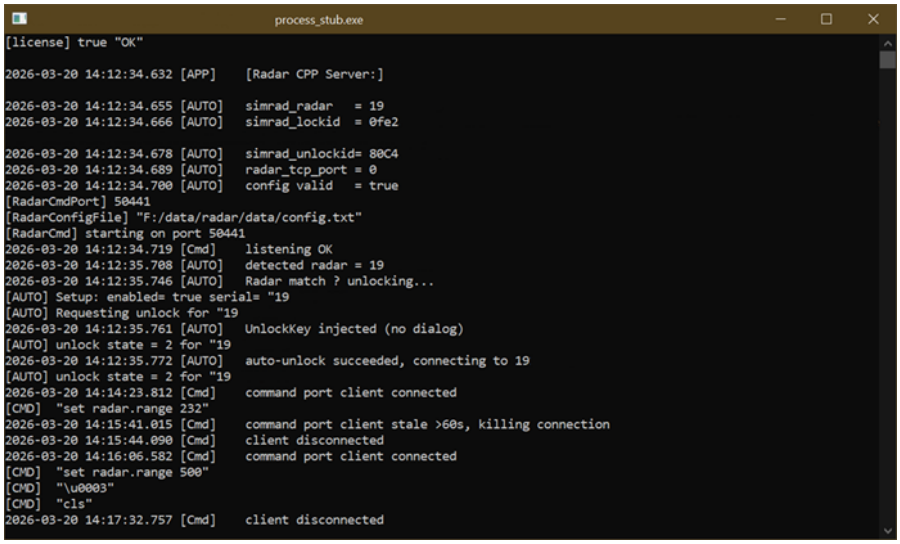
The radar hardware may internally quantize the requested value to the nearest supported operational radar range.

Note:

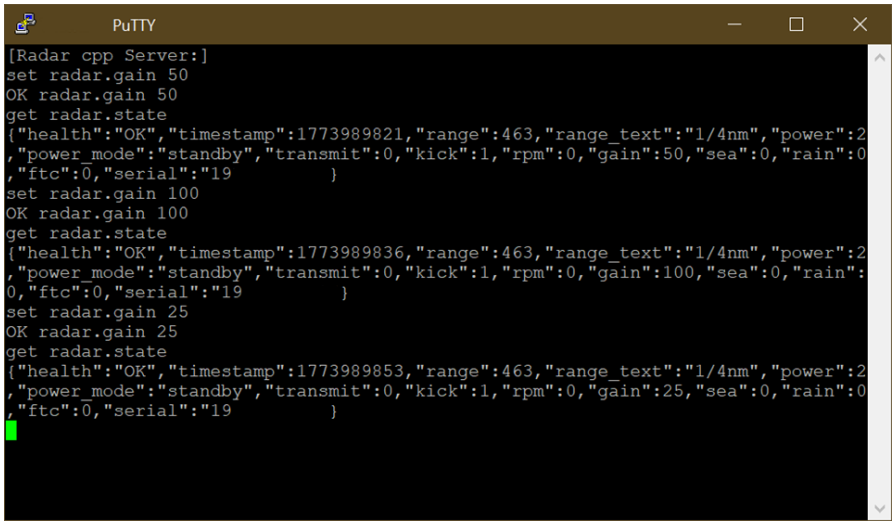
The TCP Command Port is not intended to remain permanently connected under idle conditions.

If no runtime activity is detected on the command connection for an extended period, the app may automatically close the inactive TCP session to conserve runtime resources, reduce unnecessary processing overhead, and minimize idle socket activity.

Downstream systems should therefore be designed to reconnect automatically when issuing new radar-service commands or runtime queries.



(4) *set radar.gain <nn>* [replies in Text]



```
[Radar cpp Server:]

set radar.gain 50
OK radar.gain 50

get radar.state
{"health":"OK","timestamp":1773989821,"range":463,
"range_text":"1/4nm","power":2,"power_mode":"standby",
"transmit":0,"kick":1,"rpm":0,"gain":50,"sea":0,
"rain":0,"ftc":0,"serial":"19"}

set radar.gain 100
OK radar.gain 100

get radar.state
{"health":"OK","timestamp":1773989836,"range":463,
"range_text":"1/4nm","power":2,"power_mode":"standby",
"transmit":0,"kick":1,"rpm":0,"gain":100,"sea":0,
"rain":0,"ftc":0,"serial":"19"}

set radar.gain 25
OK radar.gain 25

get radar.state
{"health":"OK","timestamp":1773989853,"range":463,
"range_text":"1/4nm","power":2,"power_mode":"standby",
"transmit":0,"kick":1,"rpm":0,"gain":25,"sea":0,
"rain":0,"ftc":0,"serial":"19"}
```

Sets the operational radar gain level.

Typical usage:

`set radar.gain 50`

Typical text reply:

`OK radar.gain 50`

The current gain value can later be verified using:

`get radar.state`

Example JSON response:

`"gain":50`

The gain value controls radar echo amplification sensitivity during spoke acquisition and radar image generation.

Higher gain values typically increase radar echo visibility but may also increase visible noise and clutter under certain operating conditions.

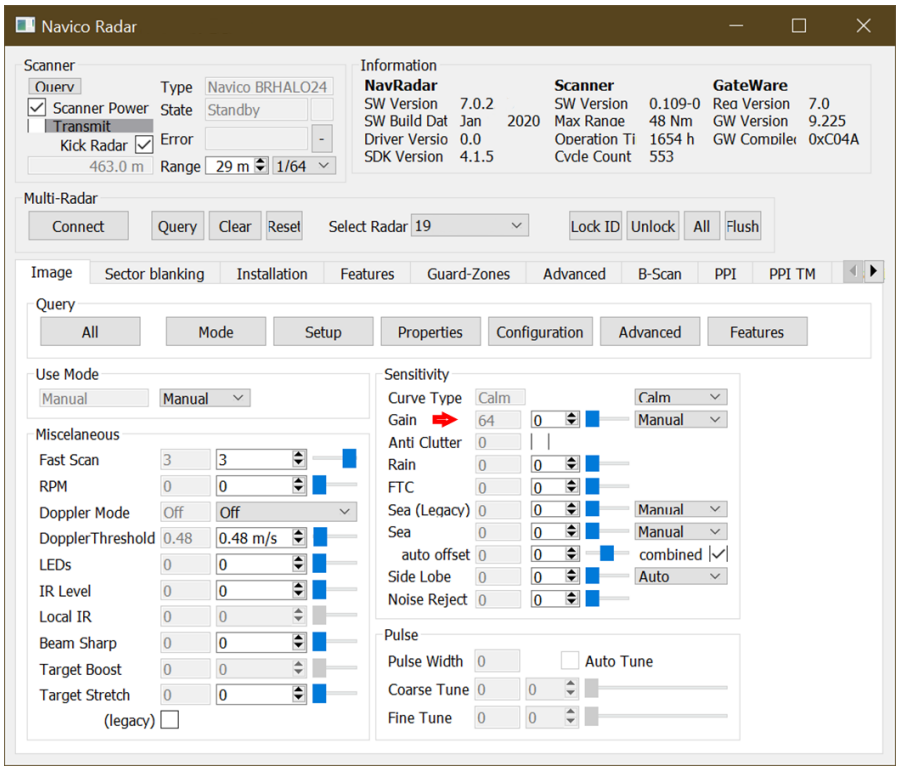
Lower gain values reduce background noise but may suppress weaker radar returns.

Typical operational range:

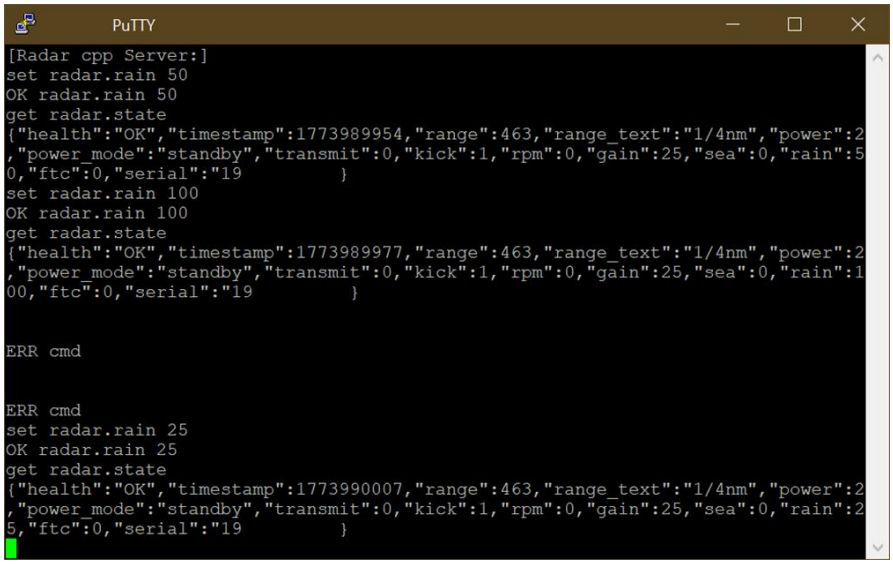
0 → minimum gain
100 → maximum gain

Gain adjustments affect:

- radar echo intensity,
- spoke acquisition visibility,
- clutter perception,
- and generated radar imagery behaviour during runtime operation.



(5) *set radar.rain <nn>* [replies in Text]



[Radar cpp Server:]

set radar.rain 50
OK radar.rain 50

get radar.state
{"health":"OK","timestamp":1773989954,"range":463,
"range_text":"1/4nm","power":2,"power_mode":"standby",
"transmit":0,"kick":1,"rpm":0,"gain":25,"sea":0,
"rain":50,"ftc":0,"serial":"19"}

set radar.rain 100
OK radar.rain 100

get radar.state
{"health":"OK","timestamp":1773989977,"range":463,
"range_text":"1/4nm","power":2,"power_mode":"standby",
"transmit":0,"kick":1,"rpm":0,"gain":25,"sea":0,
"rain":100,"ftc":0,"serial":"19"}

ERR cmd

ERR cmd

set radar.rain 25
OK radar.rain 25

get radar.state
{"health":"OK","timestamp":1773990007,"range":463,
"range_text":"1/4nm","power":2,"power_mode":"standby",
"transmit":0,"kick":1,"rpm":0,"gain":25,"sea":0,
"rain":25,"ftc":0,"serial":"19"}

Sets the operational radar rain-clutter suppression level.

Typical usage:

```
set radar.rain 50
```

Typical text reply:

```
OK radar.rain 50
```

The current rain-clutter setting can later be verified using:

```
get radar.state
```

Example JSON response:

```
"rain":50
```

The rain-clutter value controls radar filtering behaviour intended to suppress precipitation-related radar noise during spoke acquisition and radar image generation.

Higher rain values typically increase rain-clutter suppression but may also reduce visibility of weaker radar returns under certain operating conditions.

Lower rain values preserve more raw radar echo behaviour but may increase visible precipitation clutter.

Typical operational range:

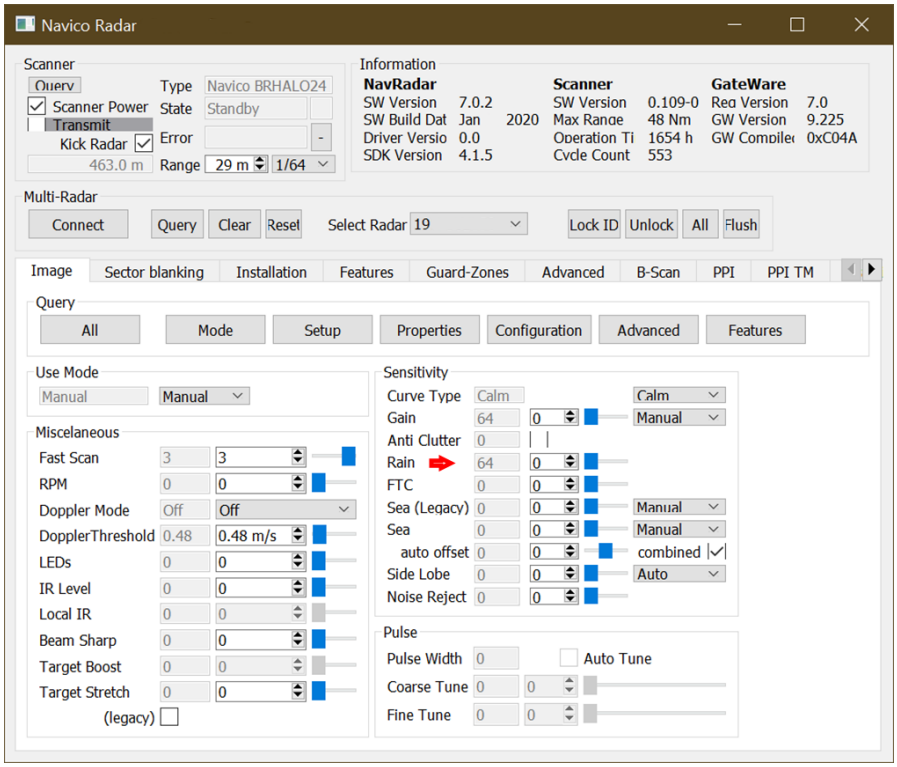
```
0 → minimum rain filtering
100 → maximum rain filtering
```

Invalid or malformed commands may produce:

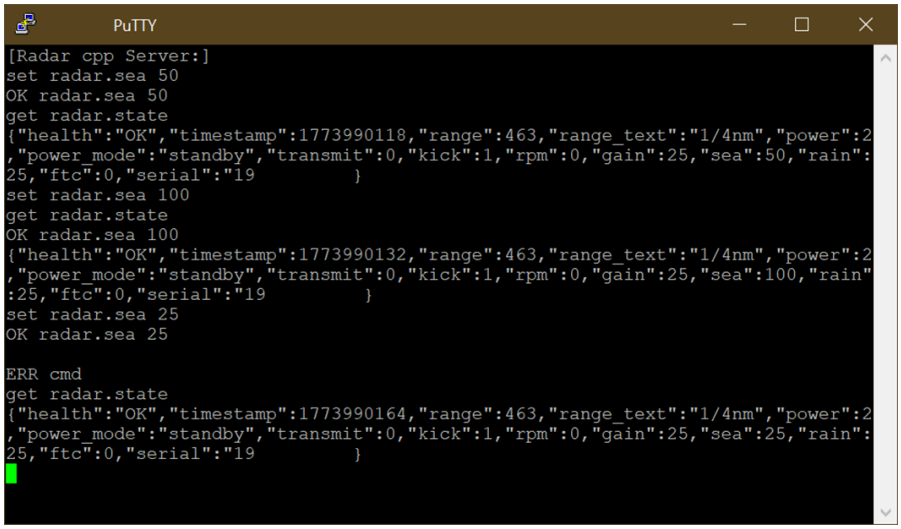
```
ERR cmd
```

Rain-clutter adjustments affect:

- radar echo filtering,
- precipitation clutter suppression,
- spoke acquisition behaviour,
- and generated radar imagery during runtime operation.



(6) *set radar.sea <nn>* [replies in Text]



[Radar cpp Server:]

set radar.sea 50
OK radar.sea 50

get radar.state
{"health":"OK","timestamp":1773990118,"range":463,
"range_text":"1/4nm","power":2,"power_mode":"standby",
"transmit":0,"kick":1,"rpm":0,"gain":25,"sea":50,
"rain":25,"ftc":0,"serial":"19"}

set radar.sea 100
OK radar.sea 100

get radar.state
{"health":"OK","timestamp":1773990132,"range":463,
"range_text":"1/4nm","power":2,"power_mode":"standby",
"transmit":0,"kick":1,"rpm":0,"gain":25,"sea":100,
"rain":25,"ftc":0,"serial":"19"}

set radar.sea 25
OK radar.sea 25

ERR cmd

get radar.state
{"health":"OK","timestamp":1773990164,"range":463,
"range_text":"1/4nm","power":2,"power_mode":"standby",
"transmit":0,"kick":1,"rpm":0,"gain":25,"sea":25,
"rain":25,"ftc":0,"serial":"19"}

Sets the operational radar sea-clutter suppression level.

Typical usage:

set radar.sea 50

Typical text reply:

```
OK radar.sea 50
```

The current sea-clutter setting can later be verified using:

```
get radar.state
```

Example JSON response:

```
"sea":50
```

The sea-clutter value controls radar filtering behaviour intended to suppress short-range sea-surface reflections and wave clutter during spoke acquisition and radar image generation.

Higher sea values typically increase sea-clutter suppression but may also reduce visibility of weaker nearby radar returns under certain sea conditions.

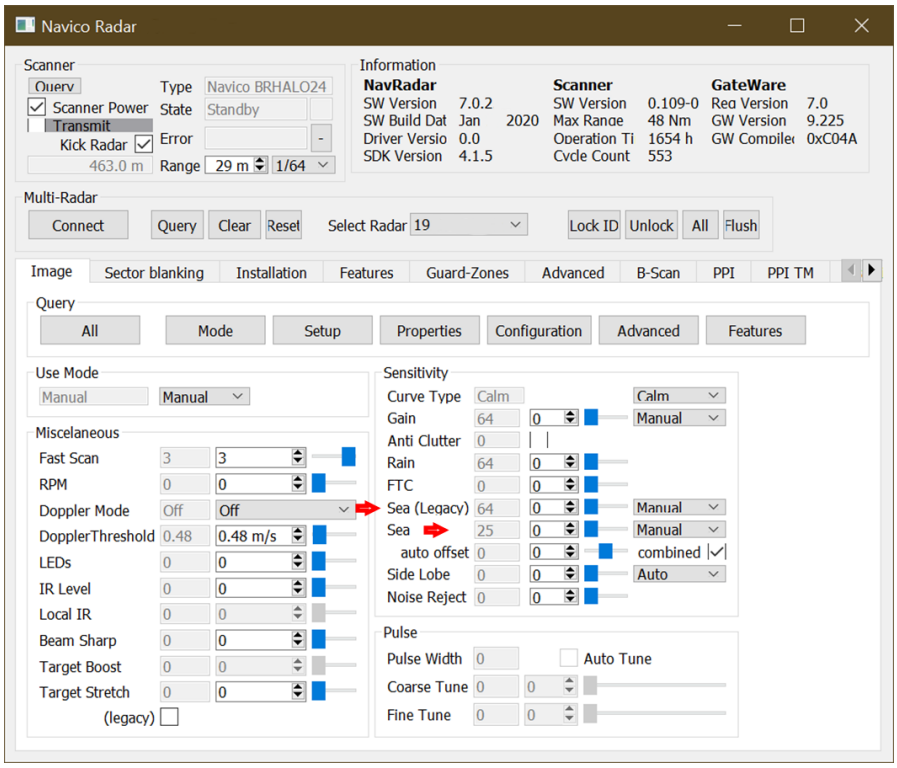
Lower sea values preserve more raw radar echo behaviour but may increase visible wave and sea-surface clutter.

Typical operational range:

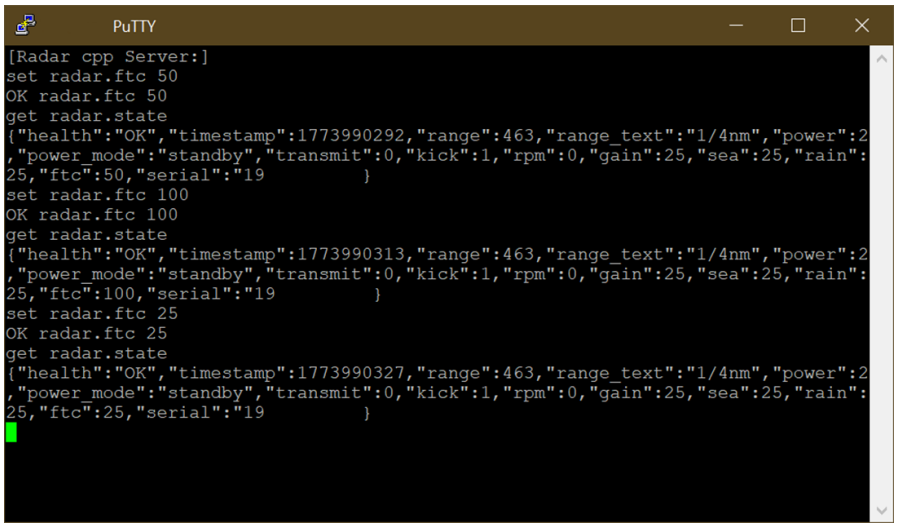
```
0 → minimum sea filtering
100 → maximum sea filtering
```

Sea-clutter adjustments affect:

- short-range radar echo behaviour,
- sea-surface clutter suppression,
- spoke acquisition behaviour,
- and generated radar imagery during runtime operation.



(7) *set radar.ftc <nn>* [replies in Text]



```
[Radar cpp Server:]

set radar.ftc 50
OK radar.ftc 50

get radar.state
{"health":"OK","timestamp":1773990292,"range":463,
"range_text":"1/4nm","power":2,"power_mode":"standby",
"transmit":0,"kick":1,"rpm":0,"gain":25,"sea":25,
"rain":25,"ftc":50,"serial":"19"}

set radar.ftc 100
OK radar.ftc 100

get radar.state
{"health":"OK","timestamp":1773990313,"range":463,
"range_text":"1/4nm","power":2,"power_mode":"standby",
"transmit":0,"kick":1,"rpm":0,"gain":25,"sea":25,
"rain":25,"ftc":100,"serial":"19"}

set radar.ftc 25
OK radar.ftc 25

get radar.state
{"health":"OK","timestamp":1773990327,"range":463,
"range_text":"1/4nm","power":2,"power_mode":"standby",
"transmit":0,"kick":1,"rpm":0,"gain":25,"sea":25,
"rain":25,"ftc":25,"serial":"19"}
```

Sets the operational radar FTC (Fast Time Constant) filtering level.

Typical usage:

set radar.ftc 50

Typical text reply:

OK radar.ftc 50

The current FTC value can later be verified using:

get radar.state

Example JSON response:

"ftc":50

The FTC value controls radar filtering behaviour intended to suppress extended-range clutter and rain-related echo smearing during spoke acquisition and radar image generation.

Higher FTC values typically increase long-echo suppression but may also reduce persistence of weaker radar returns under certain operating conditions.

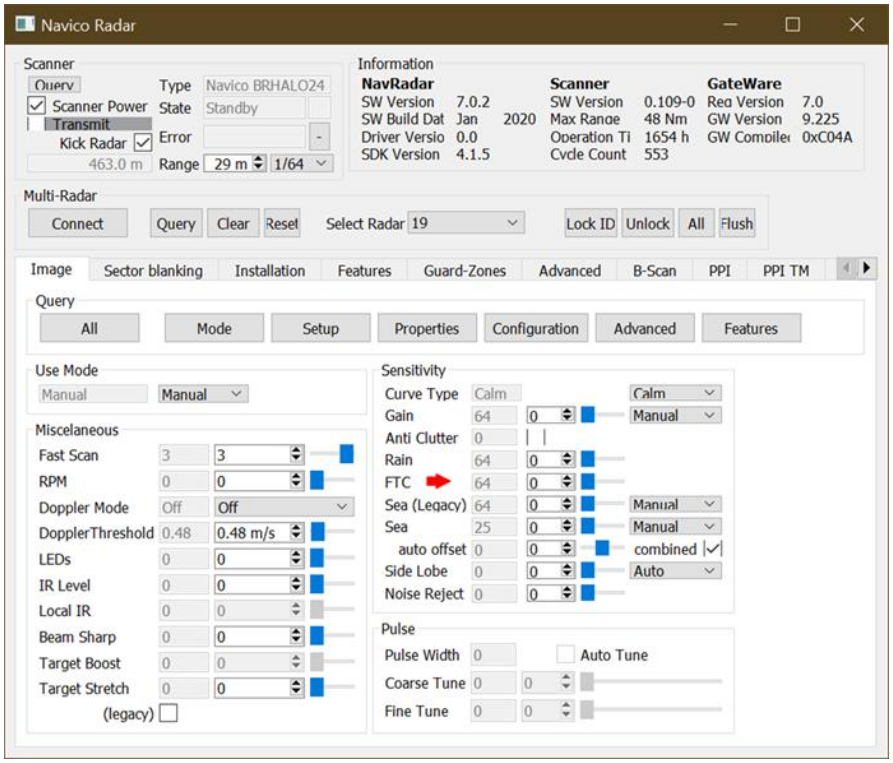
Lower FTC values preserve more raw radar echo behaviour but may increase visible streaking or clutter persistence.

Typical operational range:

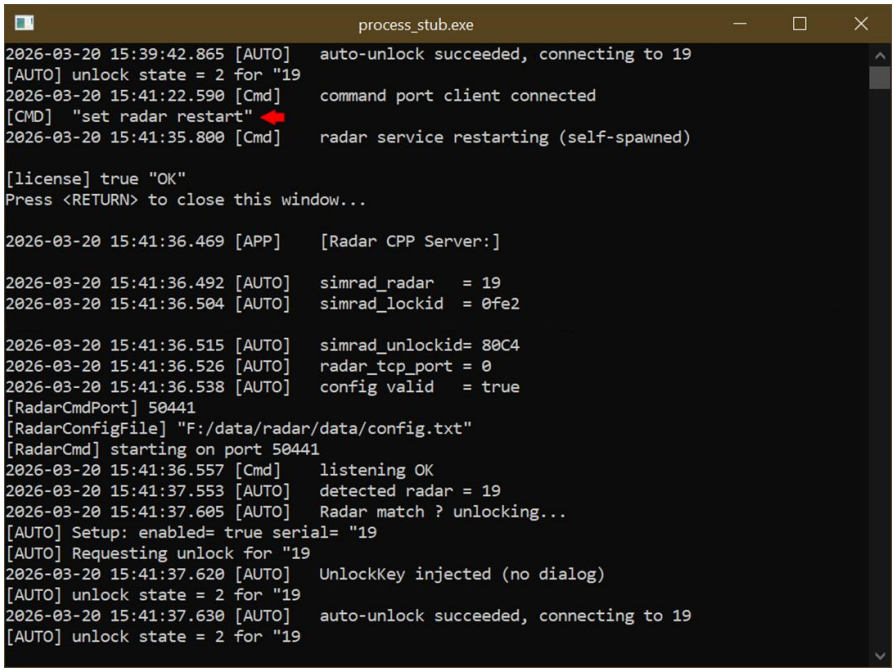
0 → minimum FTC filtering
100 → maximum FTC filtering

FTC adjustments affect:

- long-range clutter suppression,
- echo persistence behaviour,
- spoke acquisition characteristics,
- and generated radar imagery during runtime operation.



(8) *set radar restart* [replies in Text]



2026-03-20 15:39:42.865 [AUTO] auto-unlock succeeded, connecting to 19
[AUTO] unlock state = 2 for "19"

2026-03-20 15:41:22.590 [Cmd] command port client connected

[CMD] set radar restart

2026-03-20 15:41:35.800 [Cmd] radar service restarting (self-spawned)

[license] true "OK"

Press <RETURN> to close this window...

2026-03-20 15:41:36.469 [APP] [Radar CPP Server:]

2026-03-20 15:41:36.492 [AUTO] simrad_radar = 19
2026-03-20 15:41:36.504 [AUTO] simrad_lockid = 0fe2

2026-03-20 15:41:36.515 [AUTO] simrad_unlockid = 80C4
2026-03-20 15:41:36.526 [AUTO] radar_tcp_port = 0
2026-03-20 15:41:36.538 [AUTO] config valid = true

[RadarCmdPort] 50441
[RadarConfigFile] "F:/data/radar/data/config.txt"

[RadarCmd] starting on port 50441

2026-03-20 15:41:36.557 [Cmd] listening OK

2026-03-20 15:41:37.553 [AUTO] detected radar = 19
2026-03-20 15:41:37.605 [AUTO] Radar match ? unlocking...

[AUTO] Setup: enabled=true serial="19"

[AUTO] Requesting unlock for "19"

```
2026-03-20 15:41:37.620 [AUTO] UnlockKey injected (no dialog)
[AUTO] unlock state = 2 for "19"
2026-03-20 15:41:37.630 [AUTO] auto-unlock succeeded, connecting to 19
[AUTO] unlock state = 2 for "19"
```

Requests a controlled self-restart of the radar-head service runtime.

Typical usage:

```
set radar restart
```

Typical text reply:

```
OK radar restart
```

After the restart command is accepted:

- the current radar-service runtime begins shutdown,
- the radar-head process performs a controlled self-spawn restart sequence,
- and the radar acquisition pipeline is reinitialized automatically.

Typical operational console behaviour may include:

```
[Cmd] radar service restarting (self-spawned)
```

followed by:

```
[Radar CPP Server:]
```

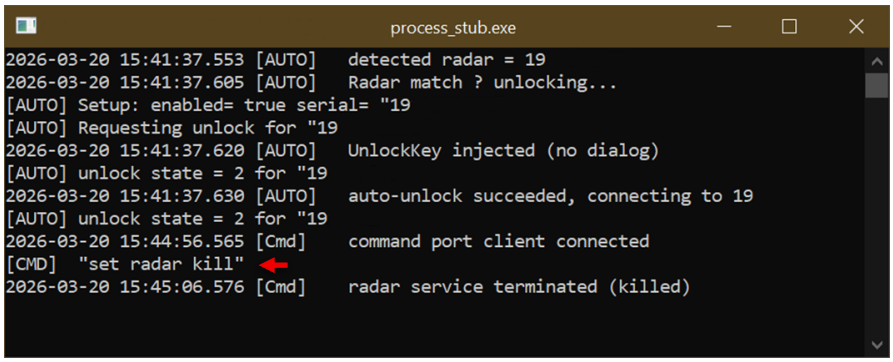
and subsequent radar startup initialization activity.

The restart mechanism is primarily intended for:

- operational recovery,
- radar-service refresh,
- replay and integration workflows,
- and fault-tolerant deployment environments.

The restart behaviour is intentionally isolated to the radar-head module so that surrounding systems do not require full-system restart during radar-service recovery operations.

(9) *set radar kill* [replies in Text]



2026-03-20 15:41:37.553 [AUTO] detected radar = 19

2026-03-20 15:41:37.605 [AUTO] Radar match ? unlocking...

[AUTO] Setup: enabled=true serial="19"

[AUTO] Requesting unlock for "19"

2026-03-20 15:41:37.620 [AUTO] UnlockKey injected (no dialog)

[AUTO] unlock state = 2 for "19"

2026-03-20 15:41:37.630 [AUTO] auto-unlock succeeded, connecting to 19

[AUTO] unlock state = 2 for "19"

2026-03-20 15:44:56.565 [Cmd] command port client connected

[CMD] "set radar kill"

2026-03-20 15:45:06.576 [Cmd] radar service terminated (killed)

Requests controlled termination of the radar-head service runtime [app-wise].

Typical usage:

set radar kill

Typical text reply:

OK radar kill

After the kill command is accepted:

- the radar-service runtime begins controlled shutdown,
- active radar acquisition is terminated,
- and the radar-head process exits cleanly.

Typical operational console behaviour may include:

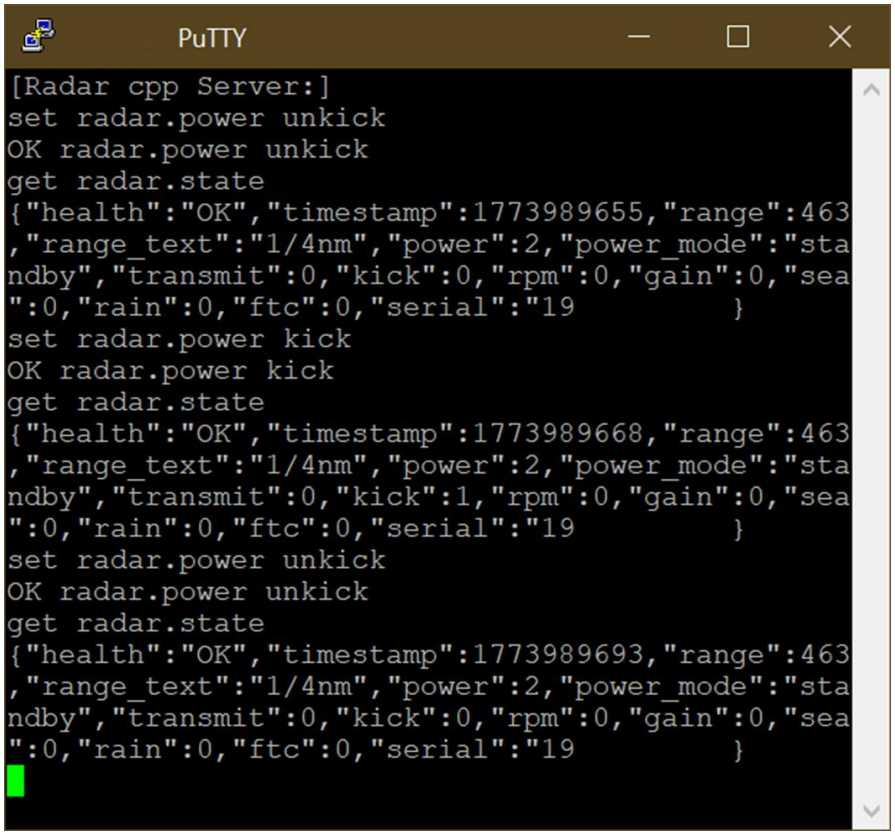
```
[Cmd] radar service terminated (killed)
```

The kill mechanism is primarily intended for:

- controlled operational shutdown,
- service maintenance,
- deployment management,
- and fault-tolerant runtime environments requiring deterministic service termination.

Unlike restart behaviour, the kill command does not automatically relaunch the radar-head service unless an external supervisor or deployment watchdog is configured to restart the process automatically.

(10) *set radar.power <kick/unkick>* [replies in Text]



[Radar cpp Server:]

set radar.power unkick
OK radar.power unkick

get radar.state
{"health":"OK","timestamp":1773989655,"range":463,
"range_text":"1/4nm","power":2,"power_mode":"standby",
"transmit":0,"kick":0,"rpm":0,"gain":0,"sea":0,
"rain":0,"ftc":0,"serial":"19"}

set radar.power kick
OK radar.power kick

get radar.state
{"health":"OK","timestamp":1773989668,"range":463,
"range_text":"1/4nm","power":2,"power_mode":"standby",
"transmit":0,"kick":1,"rpm":0,"gain":0,"sea":0,
"rain":0,"ftc":0,"serial":"19"}

set radar.power unkick
OK radar.power unkick

get radar.state
{"health":"OK","timestamp":1773989693,"range":463,
"range_text":"1/4nm","power":2,"power_mode":"standby",
"transmit":0,"kick":0,"rpm":0,"gain":0,"sea":0,
"rain":0,"ftc":0,"serial":"19"}

Controls the radar kick state used during radar operational handling and radar-service control.

Typical usage:

set radar.power kick

Typical text reply:

OK radar.power kick

or

set radar.power unkick

Typical text reply:

OK radar.power unkick

The current kick state can later be verified using:

get radar.state

where:

"kick":1

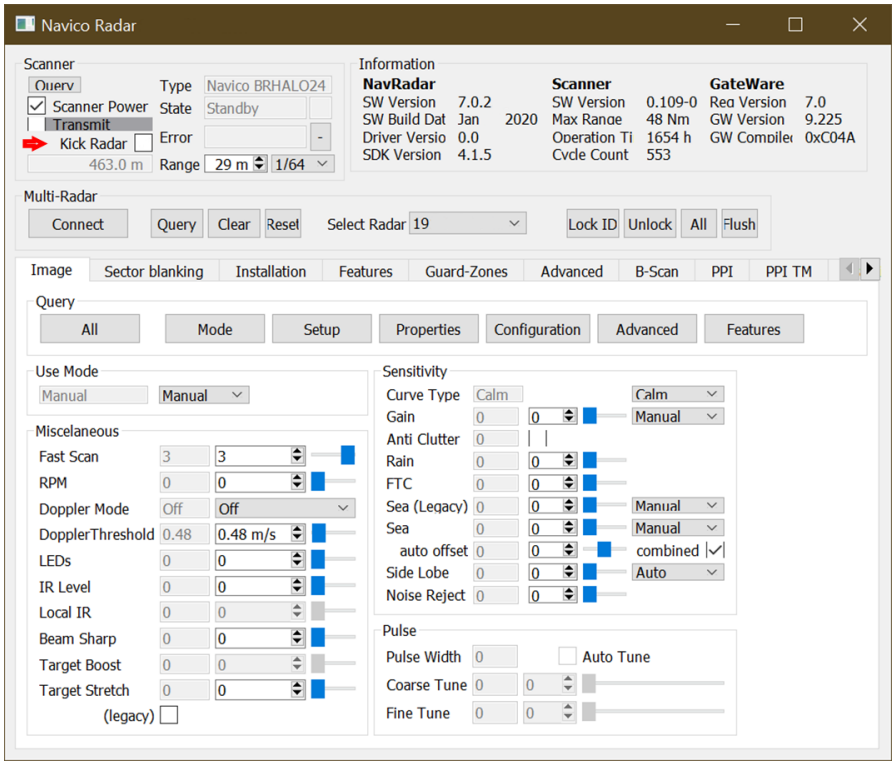
indicates the radar is currently in the kick state, while:

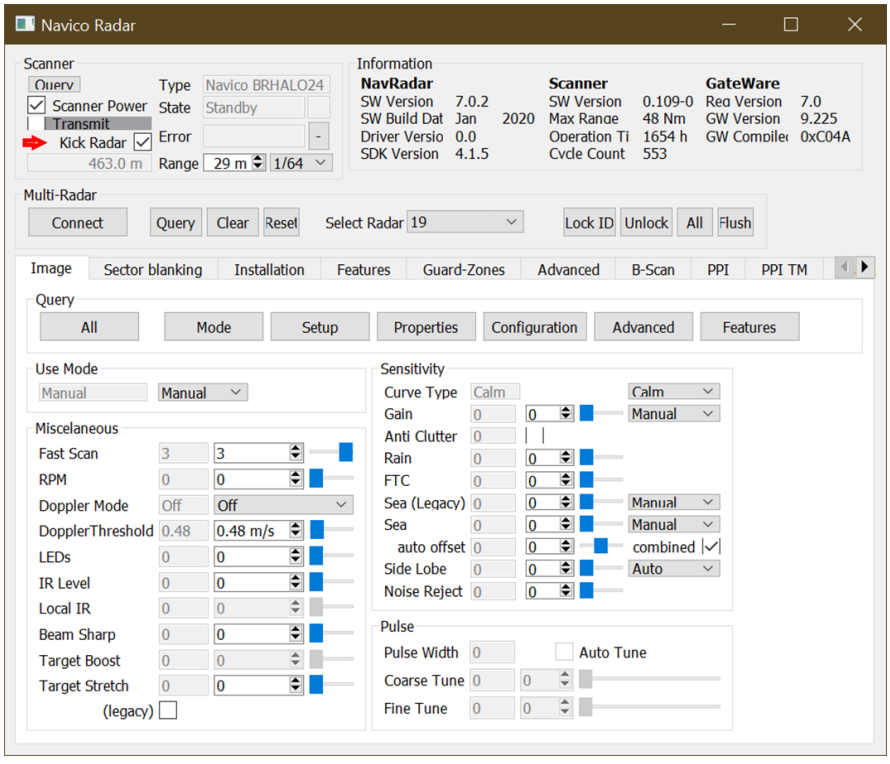
"kick":0

indicates the radar is currently in the *unkick* state.

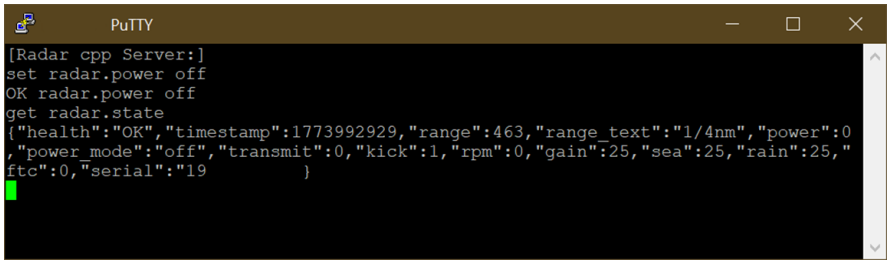
The kick mechanism is primarily used for:

- radar operational control,
- acquisition-state handling,
- startup and recovery workflows,
- and controlled radar-service behaviour during runtime operation.


Navico Radar
Scanner
Query
Scanner Power
Transmit
Kick Radar
463.0 m
Type
Navico BRHALO24
State
Standby
Error
Range
29 m
1/64
Information
NavRadar
SW Version 7.0.2
SW Build Dat Jan 2020
Driver Versio 0.0
SDK Version 4.1.5
Scanner
SW Version 0.109-0
Max Range 48 Nm
Operation Ti 1654 h
Cycle Count 553
GateWare
Req Version 7.0
GW Version 9.225
GW Compile 0xC04A
Multi-Radar
Connect
Query
Clear
Reset
Select Radar 19
Lock ID
Unlock
All
Flush
Image
Sector blanking
Installation
Features
Guard-Zones
Advanced
B-Scan
PPI
PPI TM
Query
All
Mode
Setup
Properties
Configuration
Advanced
Features
Use Mode
Manual
Manual
Miscelaneous
Fast Scan
RPM
Doppler Mode
DopplerThreshold
LEDs
IR Level
Local IR
Beam Sharp
Target Boost
Target Stretch
(legacy)
Off
0.48 m/s
Sensitivity
Curve Type
Calm
Gain
Anti Clutter
Rain
FTC
Sea (Legacy)
Sea
auto offset
Side Lobe
Noise Reject
combined
Auto
Pulse
Pulse Width
Auto Tune
Coarse Tune
Fine Tune


Navico Radar
Scanner
Query
Scanner Power
Transmit
Kick Radar
463.0 m
Type
Navico BRHALO24
State
Standby
Error
Range
29 m
1/64
Information
NavRadar
SW Version 7.0.2
SW Build Dat Jan 2020
Driver Versio 0.0
SDK Version 4.1.5
Scanner
SW Version 0.109-0
Max Range 48 Nm
Operation Ti 1654 h
Cycle Count 553
GateWare
Req Version 7.0
GW Version 9.225
GW Compile 0xC04A
Multi-Radar
Connect
Query
Clear
Reset
Select Radar 19
Lock ID
Unlock
All
Flush
Image
Sector blanking
Installation
Features
Guard-Zones
Advanced
B-Scan
PPI
PPI TM
Query
All
Mode
Setup
Properties
Configuration
Advanced
Features
Use Mode
Manual
Manual
Miscelaneous
Fast Scan
RPM
Doppler Mode
DopplerThreshold
LEDs
IR Level
Local IR
Beam Sharp
Target Boost
Target Stretch
(legacy)
Off
0.48 m/s
Sensitivity
Curve Type
Calm
Gain
Anti Clutter
Rain
FTC
Sea (Legacy)
Sea
auto offset
Side Lobe
Noise Reject
combined
Auto
Pulse
Pulse Width
Auto Tune
Coarse Tune
Fine Tune

(11) *set radar.power off* [replies in Text]



[Radar cpp Server:]

set radar.power off
OK radar.power off

get radar.state
{"health":"OK","timestamp":1773992929,"range":463,
"range_text":"1/4nm","power":0,"power_mode":"off",
"transmit":0,"kick":1,"rpm":0,"gain":25,"sea":25,
"rain":25,"ftc":0,"serial":"19"}

Requests transition of the radar hardware into the OFF power state.

Typical usage:

set radar.power off

Typical text reply:

OK radar.power off

The resulting radar power state can later be verified using:

get radar.state

Example JSON response:

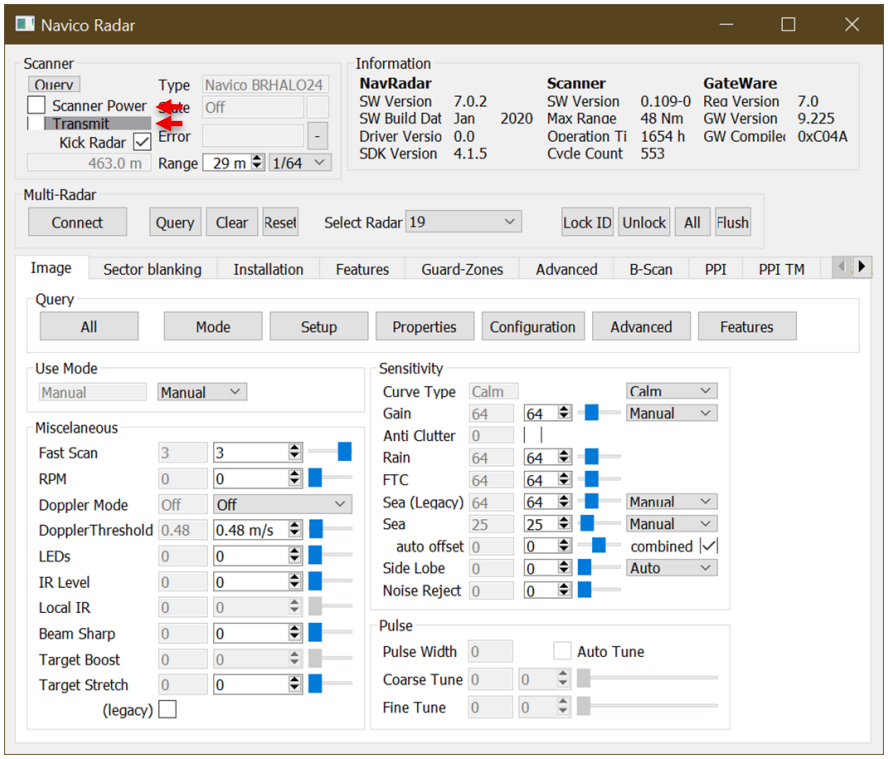
{"health":"OK","timestamp":1773992929,"range":463,
"range_text":"1/4nm","power":0,"power_mode":"off",
"transmit":0,"kick":1,"rpm":0,"gain":25,"sea":25,
"rain":25,"ftc":0,"serial":"19"}

When the radar enters the OFF state:

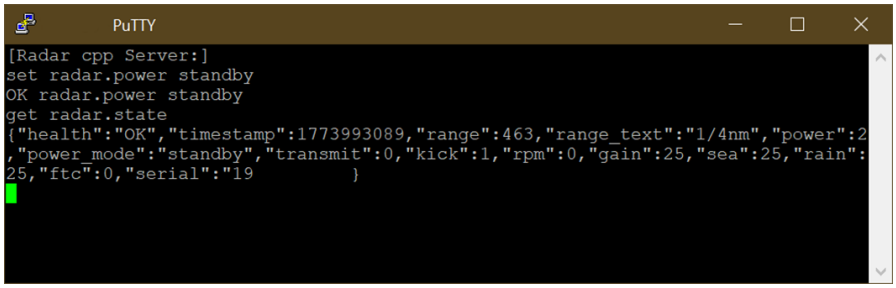
- the radar transmitter is disabled,
- spoke acquisition stops,
- sweep progression halts,
- and radar image generation is suspended until the radar is powered on again.

The OFF mode is typically used for:

- controlled radar shutdown,
- power-state management,
- maintenance workflows,
- and operational standby handling.



(12) *set radar.power standby* [replies in Text]



[Radar cpp Server:]

set radar.power standby
OK radar.power standby

get radar.state
{"health":"OK","timestamp":1773993089,"range":463,
"range_text":"1/4nm","power":2,"power_mode":"standby",
"transmit":0,"kick":1,"rpm":0,"gain":25,"sea":25,
"rain":25,"ftc":0,"serial":"19"}

Requests transition of the radar hardware into the STANDBY power state.

Typical usage:

set radar.power standby

Typical text reply:

OK radar.power standby

The resulting radar power state can later be verified using:

get radar.state

Example JSON response:

{"health":"OK","timestamp":1773993089,"range":463,
"range_text":"1/4nm","power":2,"power_mode":"standby",
"transmit":0,"kick":1,"rpm":0,"gain":25,"sea":25,
"rain":25,"ftc":0,"serial":"19"}

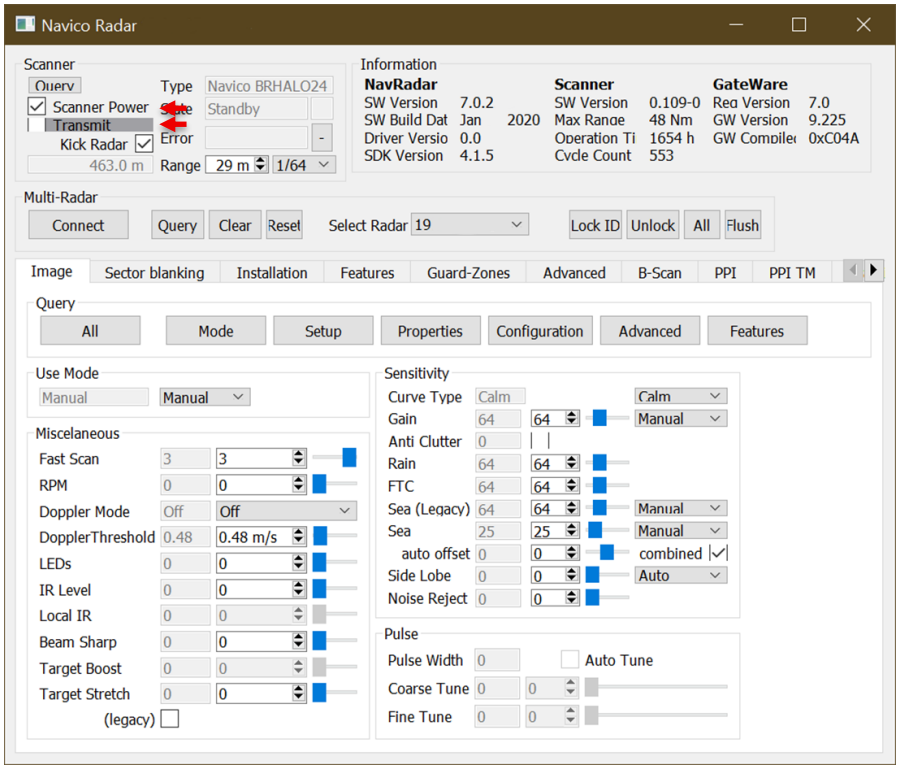
When the radar enters the STANDBY state:

- the radar hardware remains powered,
- the radar transmitter remains disabled,
- active spoke acquisition stops,
- and radar image generation is suspended until transmission resumes.

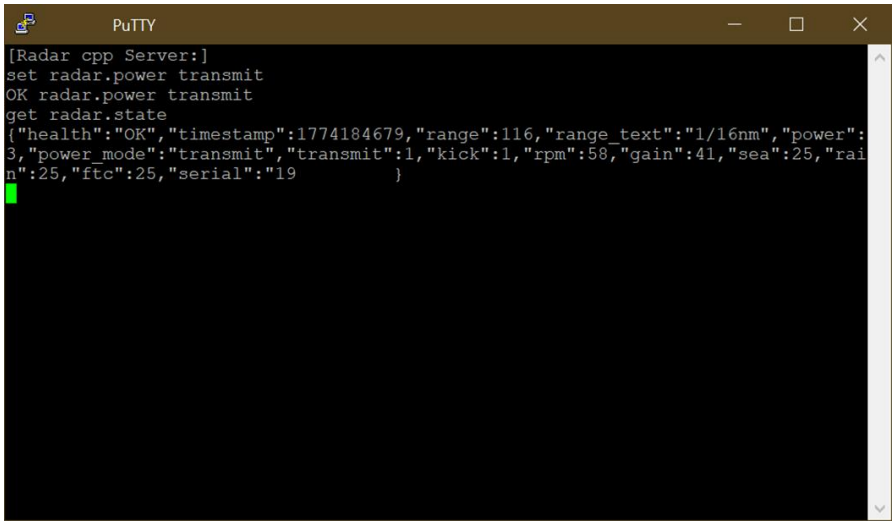
Unlike the OFF state, STANDBY mode keeps the radar system initialized and ready for faster operational transition back into transmit mode.

The STANDBY mode is commonly used for:

- temporary operational pauses,
- reduced radar transmission periods,
- and rapid transition back into live radar acquisition when required.



(13) *set radar.power transmit* [replies in Text]



[Radar cpp Server:]

set radar.power transmit
OK radar.power transmit

get radar.state
{"health":"OK","timestamp":1774184679,"range":116,
"range_text":"1/16nm","power":3,"power_mode":"transmit",
"transmit":1,"kick":1,"rpm":58,"gain":41,"sea":25,
"rain":25,"ftc":25,"serial":"19"}

Requests transition of the radar hardware into the TRANSMIT operational state.

Typical usage:

set radar.power transmit

Typical text reply:

OK radar.power transmit

The resulting radar operational state can later be verified using:

get radar.state

Example JSON response:

{"health":"OK","timestamp":1774184679,"range":116,
"range_text":"1/16nm","power":3,"power_mode":"transmit",
"transmit":1,"kick":1,"rpm":58,"gain":41,"sea":25,
"rain":25,"ftc":25,"serial":"19"}

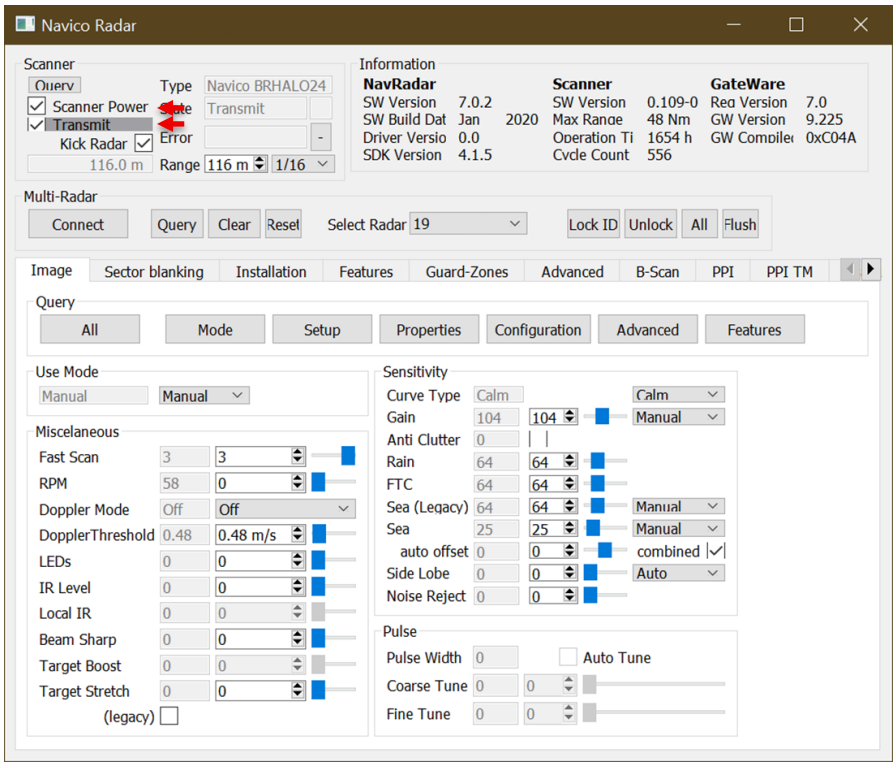
When the radar enters the TRANSMIT state:

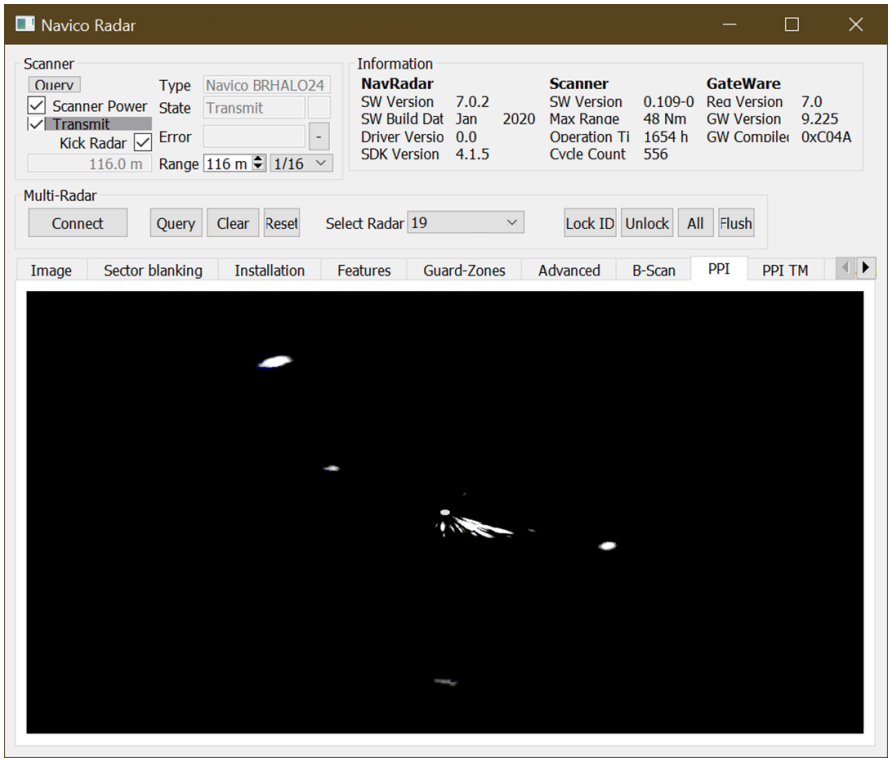
- the radar transmitter becomes active,
- live spoke acquisition begins,
- sweep progression resumes,
- and radar image generation becomes operational.

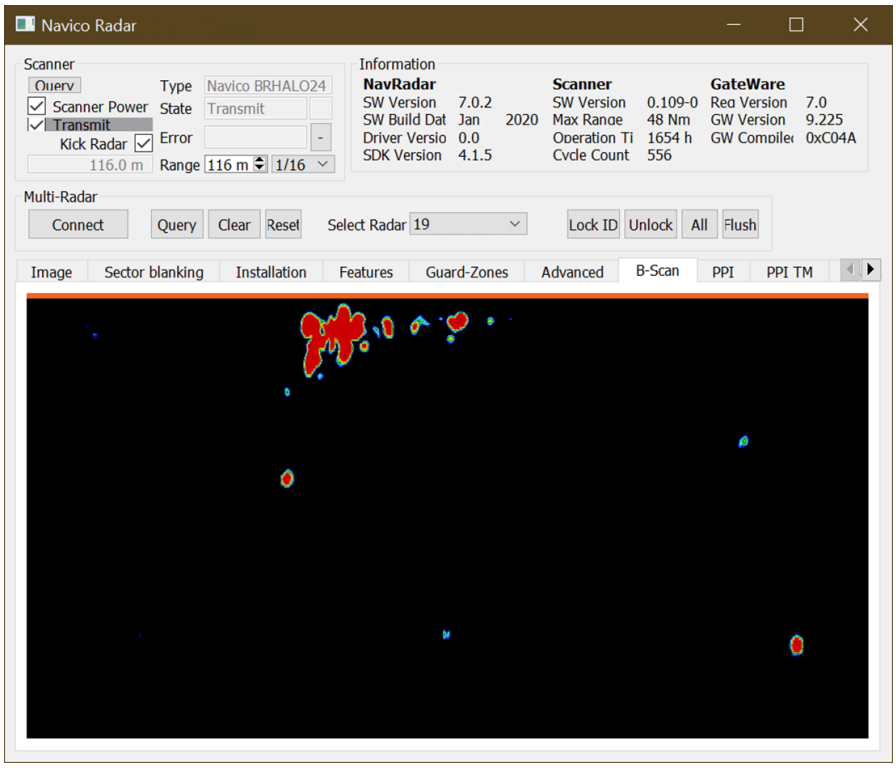
The TRANSMIT mode represents normal live radar acquisition operation where radar echoes, sweep timing, and operational radar behaviour are actively generated from the connected radar hardware.

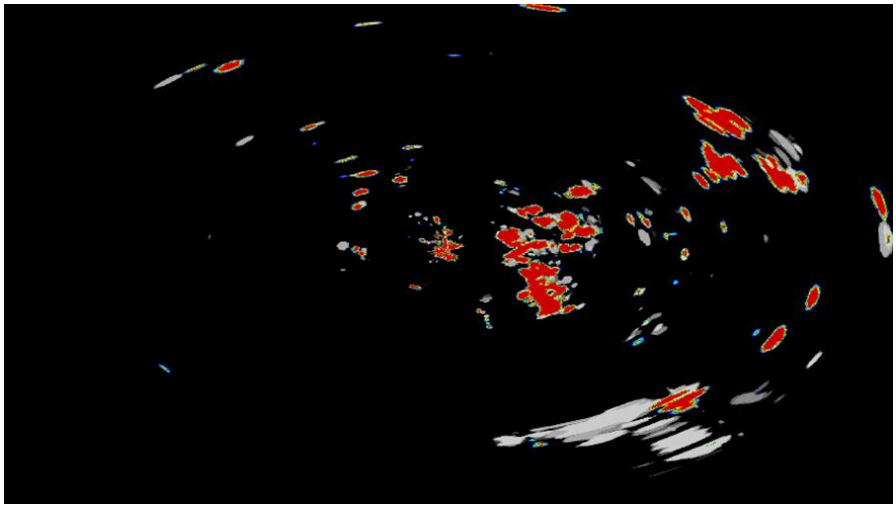
Typical operational indicators during transmit mode may include:

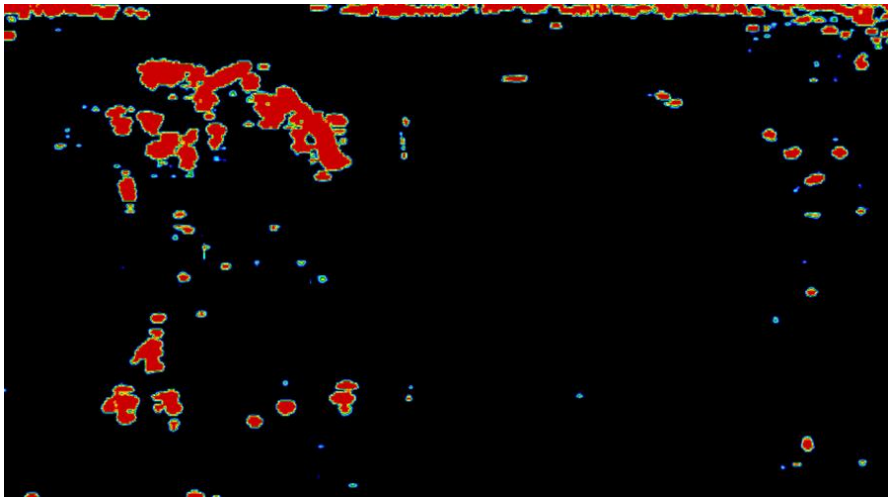
- active RPM values,
- live sweep progression,
- updating radar imagery,
- and continuous spoke acquisition activity.


Navico Radar
Scanner
Query
Scanner Power
Transmit
Kick Radar
116.0 m
Type
Navico BRHALO24
State
Transmit
Error
Range
116 m
1/16
Information
NavRadar
SW Version 7.0.2
SW Build Dat Jan 2020
Driver Versio 0.0
SDK Version 4.1.5
Scanner
SW Version 0.109-0
Max Range 48 Nm
Operation Ti 1654 h
Cycle Count 556
GateWare
Req Version 7.0
GW Version 9.225
GW Compile 0xC04A
Multi-Radar
Connect
Query
Clear
Reset
Select Radar 19
Lock ID
Unlock
All
Flush
Image
Sector blanking
Installation
Features
Guard-Zones
Advanced
B-Scan
PPI
PPI TM
Query
All
Mode
Setup
Properties
Configuration
Advanced
Features
Use Mode
Manual
Manual
Miscelaneous
Fast Scan
RPM
Doppler Mode
DopplerThreshold
LEDs
IR Level
Local IR
Beam Sharp
Target Boost
Target Stretch
(legacy)
Sensitivity
Curve Type
Gain
Anti Clutter
Rain
FTC
Sea (Legacy)
Sea
auto offset
Side Lobe
Noise Reject
Calm
Manual
combined
Auto
Pulse
Pulse Width
Auto Tune
Coarse Tune
Fine Tune


Navico Radar
Scanner
Query
Scanner Power
Transmit
Kick Radar
116.0 m
Type
Navico BRHALO24
State
Transmit
Error
Range
116 m
1/16
Information
NavRadar
SW Version 7.0.2
SW Build Dat Jan 2020
Driver Versio 0.0
SDK Version 4.1.5
Scanner
SW Version 0.109-0
Max Range 48 Nm
Operation Ti 1654 h
Cycle Count 556
GateWare
Req Version 7.0
GW Version 9.225
GW Compile 0xC04A
Multi-Radar
Connect
Query
Clear
Reset
Select Radar 19
Lock ID
Unlock
All
Flush
Image
Sector blanking
Installation
Features
Guard-Zones
Advanced
B-Scan
PPI
PPI TM


Navico Radar
Scanner
Query
Scanner Power
Transmit
Kick Radar
116.0 m
Type
Navico BRHALO24
State
Transmit
Error
Range
116 m
1/16
Information
NavRadar
SW Version 7.0.2
SW Build Dat Jan 2020
Driver Versio 0.0
SDK Version 4.1.5
Scanner
SW Version 0.109-0
Max Range 48 Nm
Operation Ti 1654 h
Cycle Count 556
GateWare
Reg Version 7.0
GW Version 9.225
GW Compile 0xC04A
Multi-Radar
Connect
Query
Clear
Reset
Select Radar 19
Lock ID
Unlock
All
Flush
Image
Sector blanking
Installation
Features
Guard-Zones
Advanced
B-Scan
PPI
PPI TM





14. Reliability & Watchdogs

The *app* is designed to cooperate with external launchers, supervisors, and fault-tolerant deployment environments through explicit runtime heartbeat and control mechanisms.

Operational liveness is exposed through heartbeat activity and controlled runtime markers such as:

- **Heartbeat** – periodically updated runtime activity used to indicate that the radar-service process remains alive and operational.
- **kill**_* markers – request controlled radar-service termination.
- **restart**_* markers – request controlled radar-service restart through the supervising launcher or runtime environment.
- **freeze**_*/stale markers – used to simulate or trigger controlled stale-heartbeat and degraded-runtime behaviour during operational testing and recovery validation.

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.

15. Known Limitations

- Different radar hardware families, SDK versions, transport behaviours, and antenna generations may exhibit different operational quirks and runtime characteristics.

- The radar-head module is therefore designed to reduce reliance on vendor-specific behaviour as much as reasonably possible, but some radar operational differences remain unavoidable across different deployment environments.

- The TCP command port currently does not implement authentication or encrypted transport security.

- Deployment is therefore recommended only within trusted operational environments, isolated networks, VPN-protected infrastructure, or behind appropriate firewall protection.

- The operational behaviour described in this document corresponds to ***`App_Radar v1.04`***.

- While runtime behaviour and supported radar capabilities may evolve across future releases, the lightweight ***`GET/CMD/SET`*** operational command philosophy is expected to remain operationally stable for downstream integrations when reasonably practical.

- Each radar-head module for differently branded radar heads has their own quirks. But we have tried several radar types to make this into something less reliant on particular radar oem behaviour.

16. 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.

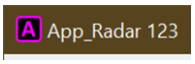
You may replace the file [***app_radar.ico***] (please see below) with any icon that represents your own brand. When the application runs, this icon will be used in the title bar and taskbar.

On a fully *unlocked* ***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>:\data\radar\data\app_radar.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>:\data\radar\data\app_radar_splash.png
```

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

At runtime, the app will display your splash logo for approximately 0.5–2.5 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."***

17. 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**:

Code:

```
<drive:>/
└──Apps/
    └──App_Radar_Group/
        └──App_Radar/
            ├── App_Radar.exe
            ├── …
            └── data.sample/  <- placeholders
                ├── _delete_.bat
                ├── app_radar.ico
                ├── app_radar_logo.png
                ├── config.txt
                └── Log_Radar_{date}_nn.txt
```

Data:

```
<drive:>/
└──data/
    └──radar/
        ├── {date..1}/
        ├── {date..2}/
        ├── …
        └──data/
            ├── app_radar.ico
            ├── app_radar_logo.png
            ├── config.txt
            └── Log_Radar_{date}_nn.txt
```

18. 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.

19. Intended Audience

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

20. 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.

21. The Math Behind the Radar Image Construction

(a) Preliminary Note

This section is currently a placeholder and operational overview rather than the final mathematical write-up.

The full mathematical derivation, reconstruction equations, timing models, spoke interpolation logic, and deterministic replay formulation are still being reconstructed and formalized for public exposure.

The intention of this placeholder is therefore not to present the complete radar mathematics yet, but to expose the underlying engineering philosophy, reconstruction direction, and operational transparency goals behind the radar-image construction pipeline.

The full mathematical write-up and detailed derivations are expected to follow in subsequent revisions once the formulations, reconstruction models, and operational explanations have been fully evaluated and consolidated.

(b) Construction of the Radar Image

The construction of the radar image is one of the most critical stages within the entire radar-service pipeline.

Before any downstream interpretation, AI-assisted deduction, object extraction, track formation, or operational reasoning can occur, the radar spoke stream must first be transformed into a mathematically coherent spatial representation of observed radar energy across time and angle.

This stage is intentionally treated as a deterministic engineering process rather than a cosmetic rendering process.

The generated radar image is not considered a “display artifact.”

It is considered an operational reconstruction of observed radar behaviour.

The quality, timing, continuity, and determinism of this reconstruction directly affect every downstream capability, including:

- echo-strength interpretation,
- target persistence analysis,
- motion estimation,
- sweep-to-sweep correlation,
- track extraction,
- clutter suppression,
- and future AI-assisted radar reasoning.

A mathematically weak radar-image construction layer will inevitably contaminate all downstream interpretation layers regardless of how sophisticated later AI or tracking systems may appear.

For this reason, the radar image construction process is intentionally designed to remain:

- deterministic,
- replay-able,
- operationally observable,
- and mathematically explainable.

(1) Radar Spokes as the Fundamental Primitive

The app fundamentally treats incoming radar spokes as the primary radar observation primitive.

Each spoke represents:

- a sweep angle,
- a time-localized radar observation,
- and a sampled sequence of radar echo intensities across radial distance bins.

Conceptually:

$spoke(angle) = [e_0, e_1, e_2, e_3, \ldots e_n]$

where:

- angle represents instantaneous antenna orientation, and
- e_n represents measured radar echo intensity at distance bin *n*.

The radar image is therefore not painted directly.

It is reconstructed progressively from thousands of sequential radial observations across time.

(2) Polar-to-Cartesian Reconstruction

Radar spokes naturally exist in polar space:

- angle,
- range,
- and echo intensity.

However, operational radar imagery is typically rendered into Cartesian image space for visualization, replay, storage, and downstream processing.

The reconstruction therefore performs continuous:

- polar-to-Cartesian projection,
- range normalization,
- and angular sweep interpolation.

Conceptually:

$$x = r\,cos(\theta), \qquad y = r\,sin(\theta)$$

where:

- r represents radar range distance, and
- θ represents spoke angle.

Each radar echo sample is therefore mapped into image-space coordinates during sweep construction.

This process must remain:

- consistent across sweep cycles,
- timing-stable,
- and operationally deterministic.

Otherwise:

- echo smearing,
- sweep tearing,
- or false target persistence

can emerge artificially from reconstruction instability rather than from actual radar observations.

(3) Sweep Continuity and Temporal Coherence

A radar image is never truly “static.”

It is continuously reconstructed from time-progressive spoke arrivals.

Therefore, image correctness depends not only on spatial math, but also on temporal coherence.

The app continuously evaluates:

- spoke ordering,
- angular continuity,
- sweep completion state,
- spoke timing intervals,
- and sweep freshness.

This becomes operationally important because:

- missing spokes,
- delayed transport,
- burst spoke arrival,
- or stale replay buffers

can produce visually plausible yet mathematically corrupted radar imagery.

The app therefore treats timing behaviour as part of the image-construction math itself rather than as a separate transport concern.

(4) Echo Strength Interpretation

Radar echo intensity is not treated as an absolute truth value.

Instead, the radar-service layer treats echo intensity as:

- an observed energy response,
- subject to operational conditions,
- filtering behaviour,
- and acquisition-state consistency.

Operational radar controls such as:

- gain,
- sea clutter suppression,
- rain filtering,
- and FTC behaviour

directly influence the observed echo field.

As a result, the app intentionally avoids pretending that a bright radar return automatically implies a strong physical target.

Instead, radar image construction preserves enough operational context that downstream systems may later reason about:

- echo persistence,
- clutter probability,
- sweep consistency,
- and acquisition stability.

(5) Deterministic Replay-ability

One of the primary design goals of the radar-image construction pipeline is deterministic replay-ability.

Given:

- the same spoke stream,
- the same timing behaviour,
- and the same operational radar settings,

the generated radar image sequence should reconstruct identically across replay sessions.

This is operationally important because radar anomalies are often:

- transient,
- difficult to reproduce,
- and timing-sensitive.

Replay determinism therefore allows:

- forensic debugging,
- algorithm validation,
- AI training reproducibility,
- and operational regression testing

without requiring repeated sea trials.

(6) Why This Section Matters

Many radar systems expose only the final rendered image while hiding the reconstruction assumptions that produced it.

This app intentionally moves in the opposite direction.

The long-term objective of this section is to expose the mathematical and operational reconstruction process transparently so that:

- the radar-image pipeline remains explainable,
- the reconstruction assumptions remain observable,
- and downstream AI reasoning is built on deterministic operational foundations rather than opaque visual artifacts.

The goal is therefore not merely image rendering.

The goal is mathematically honest radar reconstruction.

(7) Future Expansion

This section currently serves as an operational overview of the radar-image construction philosophy.

Future revisions are expected to expose additional mathematical detail regarding:

- spoke interpolation,
- angular quantization,
- range-bin scaling,
- echo normalization,
- sweep stitching,
- target persistence modelling,
- clutter filtering behaviour,
- and deterministic replay reconstruction mathematics.

The intention is to expose these mechanisms explicitly and transparently so that the radar reconstruction process remains

operationally understandable rather than hidden behind proprietary or unexplained rendering behaviour.

(8) On “Sleight-of-Hand” Rendering

Historically, many radar-rendering systems — especially early-stage prototypes and demonstration systems — have relied on various forms of rendering “sleight-of-hand” to improve visual appearance.

These techniques may include:

- echo persistence exaggeration,
- artificial smoothing,
- spoke interpolation beyond observable data,
- frame blending,
- latency masking,
- clutter beautification,
- or visually stabilizing unstable radar behaviour.

In moderation, some of these techniques may improve operator readability.

However, excessive visual manipulation can also conceal important operational truths such as:

- stale spoke behaviour,
- transport jitter,
- acquisition interruptions,
- sweep inconsistency,
- or degraded radar timing behaviour.

In practice, visually “beautiful” radar imagery is not always operationally honest radar imagery.

This is not written as criticism.

In fact, the author openly acknowledges that earlier generations of experimental radar-rendering work — including some internal prototype systems — also relied heavily on such visual stabilization approaches during the learning process.

This experience ultimately shaped an important engineering realization:

> A radar image should not merely look stable.

> It should be mathematically explainable and operationally truthful.

For this reason, the design philosophy of *App_Radar* intentionally prioritizes:

- deterministic reconstruction,
- observable acquisition behaviour,
- timing transparency,
- and replayable operational correctness

over purely cosmetic rendering smoothness.

The goal is therefore not to produce the most visually flattering radar image.

The goal is to produce a radar image whose behaviour remains explainable, reproducible, and operationally trustworthy under both nominal and degraded conditions.

22. FAQs

(c) 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.

(d) 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 app 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 app automatically blocks outputs to prevent acting on stale or misleading information.

(e) 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.

(f) 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.

(g) 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 *white-label* 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.

(h) 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.

(i) 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.

(j) 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).

(k) 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.

(l) 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.

(m) 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).

(n) 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.

(o) 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.

(p) 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 behaviour.

(q) 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).

(r) 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).

(s) 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).

(t) 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 and/or website 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.