
Table of Contents

---

1. Overview

`RCC_DashboardInputs` is responsible for **displaying dashboard information** in Realistic Car Controller (RCC)–based projects. It reads parameters from the assigned (or automatically detected) vehicle and updates UI elements such as:

- **RPM needle**
- **Speedometer needle** (KMH/MPH)
- **Turbo** gauge and needle
- **Nitrous Oxide (NoS)** gauge and needle
- **Heat** gauge and needle
- **Fuel** gauge and needle

Additionally, it provides hooks for **lighting indicators** (headlights, turn signals) and other driver-assist systems like ABS, ESP, and parking brake.

---

2. Class Declaration

```
public class RCC_DashboardInputs : RCC_Core {
    // Implementation details...
}
```

This class inherits from `RCC_Core`, which is a base class in the RCC framework. It's typically placed on a dashboard UI prefab or directly in a scene UI that needs to track vehicle stats.

---

3. Fields and Properties

This section provides an overview of the public and private fields used by `RCC_DashboardInputs`.

3.1 Vehicle References

- **`public RCC_CarControllerV4 vehicle`**
  The RCC vehicle whose data will be displayed on the dashboard.

- **`public bool autoAssignVehicle = true`**

  - When `true`, automatically fetches the current player vehicle (`RCCSceneManager.activePlayerVehicle`).
  - If you want to manually assign `vehicle`, set this to `false`.

3.2 Needle Objects and Gauges

- **Needle References**

  - **`public GameObject RPMNeedle`**
  - **`public GameObject KMHNeedle`**
  - **`public GameObject turboNeedle`**
  - **`public GameObject NoSNeedle`**
  - **`public GameObject heatNeedle`**
  - **`public GameObject fuelNeedle`**
    These are UI objects (e.g., Image, GameObject) rotated to display values on the dashboard.

- **Gauges**

  - **`public GameObject turboGauge`**
  - **`public GameObject NOSGauge`**
  - **`public GameObject heatGauge`**
  - **`public GameObject fuelGauge`**
    These parent objects will be enabled or disabled based on whether the vehicle uses turbo, NoS, engine heat, or fuel consumption.

3.3 Rotation Variables

- **`private float RPMNeedleRotation = 0f`**

- `private float KMHNeedleRotation = 0f`
- `private float BoostNeedleRotation = 0f`
- `private float NoSNeedleRotation = 0f`
- `private float heatNeedleRotation = 0f`
- `private float fuelNeedleRotation = 0f`

These floats keep track of the **current rotation** values for each needle.

3.4 Runtime Vehicle Stats

The script also exposes a few `[HideInInspector]` fields containing live vehicle data:

- `[HideInInspector] public float RPM;`
  Vehicle engine revolutions per minute.
- `[HideInInspector] public float KMH;`
  Vehicle speed (in KMH or MPH).
- `[HideInInspector] public int direction;`
  Direction of travel, +1 or -1.
- `[HideInInspector] public float Gear;`
  Current transmission gear.
- `[HideInInspector] public bool changingGear;`
  If the vehicle is in the process of shifting gears.
- `[HideInInspector] public bool NGear;`
  If the vehicle is in neutral gear.
- `[HideInInspector] public bool ABS;`
  Active ABS indicator.
- `[HideInInspector] public bool ESP;`
  Active ESP indicator.
- `[HideInInspector] public bool Park;`
  If the parking brake (handbrake) is engaged.
- `[HideInInspector] public bool Headlights;`
  If low beam or high beam headlights are active.
- `[HideInInspector] public RCC_CarControllerV4.IndicatorsOn indicators;`
  Turn signal state (None, Left, Right, or All).

---

4. How It Works

The `Update()` method is where the script gathers the vehicle's runtime info, toggles gauge visibility, and updates each needle's rotation.

4.1 Auto-Assignment of Vehicle

```
if (autoAssignVehicle && RCCSceneManager.activePlayerVehicle)
    vehicle = RCCSceneManager.activePlayerVehicle;
else
    vehicle = null;
```

- If `autoAssignVehicle` is `true`, it looks for `RCCSceneManager.activePlayerVehicle`.
- If no player vehicle is found, or if the scene manager doesn't have an active vehicle, `vehicle` becomes `null`.
- The script returns early if `vehicle == null` or if the vehicle can't be controlled.

4.2 Visibility of Gauges

```
// Example for NoS gauge
if (NOSGauge) {
    if (vehicle.useNOS) {
        if (!NOSGauge.activeSelf)
            NOSGauge.SetActive(true);
    } else {
        if (NOSGauge.activeSelf)
            NOSGauge.SetActive(false);
    }
}
```

The same pattern repeats for **turboGauge**, **heatGauge**, and **fuelGauge**, ensuring they are only visible if the vehicle actually supports those systems.

4.3 Retrieving and Updating Stats

Speed, RPM, and other indicators are read directly from `vehicle`:

```
RPM = vehicle.engineRPM;
KMH = vehicle.speed;
direction = vehicle.direction;
Gear = vehicle.currentGear;
changingGear = vehicle.changingGear;
NGear = vehicle.NGear;
ABS = vehicle.ABSAct;
ESP = vehicle.ESPAct;
Park = vehicle.handbrakeInput > .1f ? true : false;
Headlights = vehicle.lowBeamHeadLightsOn || vehicle.highBeamHeadLightsOn;
indicators = vehicle.indicatorsOn;
```

Then, each **needle** is rotated based on these values:

```
// Example for RPM needle
if (RPMNeedle) {
    RPMNeedleRotation = (vehicle.engineRPM / 50f);
    RPMNeedleRotation = Mathf.Clamp(RPMNeedleRotation, 0f, 180f);
    RPMNeedle.transform.eulerAngles = new Vector3(
        RPMNeedle.transform.eulerAngles.x,
        RPMNeedle.transform.eulerAngles.y,
        -RPMNeedleRotation
    );
}
```

Similarly for **KMHNeedle**, **turboNeedle**, **NoSNeedle**, **heatNeedle**, and **fuelNeedle**, but each with its own calculation or clamp:

1. **KMHNeedle**
    - Uses `vehicle.speed` but converts to MPH if `Settings.units` is `Units.MPH` (`speed * 0.62f`).
2. **turboNeedle**
    - Rotates based on `vehicle.turboBoost / 30f * 270f`.
3. **NoSNeedle**
    - Rotates based on `(vehicle.NoS / 100f) * 270f`.
4. **heatNeedle**
    - `(vehicle.engineHeat / 110f) * 270f`.
5. **fuelNeedle**
    - `(vehicle.fuelTank / vehicle.fuelTankCapacity) * 270f`.

These multipliers and divisions yield the correct rotation angles, matching the desired gauge “range” in your UI design.

---

5. Usage Notes and Best Practices

1. **Assigning Dashboard Prefabs**
    - Typically, you’ll have a UI prefab with needles, images, or transforms assigned to these public fields.
    - Make sure to hook them up correctly in the **Inspector**.
2. **Auto-Assign vs. Manual Assign**
    - If you want to track a specific AI or remote-controlled vehicle, set `autoAssignVehicle = false` and drag the vehicle reference directly into the Inspector.
    - Otherwise, letting `RCCSceneManager.activePlayerVehicle` be assigned automatically is convenient for single-player setups.

3. **Gauge Needle Ranges**
    - This script uses hard-coded ranges (e.g., 0 to 270 degrees for turbo, 0 to 180 degrees for RPM).
    - If your UI needle graphics have different min/max angles, adjust the math accordingly.
4. **Performance**
    - Rotating a handful of needles each frame is generally cheap.
    - If you have a large or complex dashboard, consider **Update** vs. **FixedUpdate** usage, or use LOD logic (only update the UI when necessary).
5. **International Units**
    - By default, `KMH` or `MPH` is set in `RCC_Settings`.
    - The script checks `Settings.units` to decide how to compute the speedometer. Make sure `RCC_Settings` is configured appropriately.
6. **Extension**
    - You can add more gauges (oil pressure, battery voltage, etc.) by replicating the same pattern of enabling or rotating UI objects based on new properties in the vehicle script.

---

6. Summary

`RCC_DashboardInputs` provides a straightforward method to **bind vehicle stats** to a **dashboard UI**. By automatically assigning or manually specifying the target `RCC_CarControllerV4`, it retrieves speed, RPM, gear states, and specialized data (turbo, nitrous, fuel, engine heat) to rotate needles and toggle the visibility of relevant gauges.

With minimal setup—just drag references to the needle objects and gauge parent objects—the script will continuously reflect changes in the vehicle's status, giving the player a polished, real-time instrument cluster for their RCC-based cars.