

SOFLAM Integration Core

A designator you carry, assemble on a tripod and operate as a gunner – publishing laser targets by code, so that missiles, turrets and aircraft from *other* mods can find them. These pages explain how such a system is built in Arma Reforger, and above all where it breaks once it runs on a dedicated server. If you build a mod that should work with it, start with [Make your mod compatible](#).

Online version: <https://docs.seebie90gaming.com/soflam/index.html>

- ◆ The SOFLAM Integration Core mod is published under the Arma Public License (APL)
- ◆ Example code is free to use ◆ Suggestions: Discord Seebie90 I Nomad – discord.gg/gtg-german-tactical-group

Contents

SOFLAM INTEGRATION CORE

Overview **CHANGED**

FOR OTHER MODDERS

Make your mod compatible **NEW**

1. Add the Core as a dependency **NEW**
2. Make a weapon follow the laser **NEW**
3. Aircraft, vehicles and turrets **NEW**
4. Report your own designator **NEW**
5. Optional integration **NEW**
6. Testing and common errors **NEW**

HOW IT IS BUILT

Target registry **CHANGED**

The designator **CHANGED**

Who may send **CHANGED**

The seeker side **CHANGED**

Night and thermal vision **CHANGED**

The visible beam **CHANGED**

A telescopic tripod **CHANGED**

Assembling and the gunner seat **CHANGED**

Dedicated-server pitfalls **CHANGED**

Integrating aircraft **CHANGED**

Building a laser designator other mods can home in on

A designator you carry, assemble on a tripod and operate as a gunner – publishing laser targets by code, so that missiles, turrets and aircraft from *other* mods can find them. These pages explain how such a system is built in Arma Reforger, and above all where it breaks once it runs on a dedicated server. If you build a mod that should work with it, start with [Make your mod compatible](#).

IN PLAIN WORDS

Techniques from a laser designator mod: a target registry that other mods read, a tripod you can assemble, a network path that survives a dedicated server, and night and thermal vision. Each page is one problem and how to solve it.

[VIEW IN THE WORKSHOP →](#)

How to read these pages

Every page is one problem that took a while to get right: what goes wrong with the obvious approach, the technique that works instead, and a short example. The examples are **excerpts from the real code** of the SOFLAM Integration Core, shortened and commented – class, method, attribute and action names are the real ones (`GTG_SOFLAM_`). They are not whole files. Where a snippet is a pattern for *your* mod, its classes start with `MyMod_` , which stands for your own prefix.

These pages describe the **published (stable) version** of the Core, the one on the [Workshop](#). A few member names in the code are German (`GetLetzteMeldung` , `GTG_SendeZielAnlegen`); they are quoted as they are, because that is what you have to type.

NOTE

Almost everything on these pages was found by running on a dedicated server. Single-player and Workbench play hide the largest class of bugs, because client and server are the same machine there and nobody has to own anything. That split is the recurring theme.

For other modders

Do you build a missile, an aircraft, a turret or a designator of your own? These pages tell you step by step what to do so that your mod works with the Core – with the real names, real prefab text from the mods that already do it, and the mistakes they made on the way.

Make your mod compatible

Start here: what do you want to build, the public surface of the Core, and which existing mod to copy.

1. Add the Core as a dependency

addon.gproj, the GUID, and what happens when the Core is missing.

2. Make a weapon follow the laser

Read the registry from your seeker, or attach the Core's seeker component.

3. Aircraft, vehicles and turrets

Code holder, cockpit dials, injector, seat names – as the AH-64 and AH-1Z mods build them.

4. Report your own designator

How a drone, a turret or any other device puts a laser point into the registry.

5. Optional integration

Work with or without the Core, without a hard dependency.

6. Testing and common errors

A dedicated-server checklist, and symptom → cause → fix with the real log lines.

How it is built

Target registry

Match by code, not by entity. A game system instead of a singleton, and an expiry for entries nobody removed.

The designator

Mark or only measure, follow a moving vehicle, and report rarely.

Who may send

A client that does not own the entity talks to nobody. A relay on the character fixes it.

The seeker side

How a missile or turret in another mod finds the point, and why mounts need several seekers.

Night and thermal vision

Per-viewer infrared visibility, optional mods without a compile dependency, and the camera-id trap.

The visible beam

A component that ticks only where the entity is owned, and why the beam must start at the lens.

A telescopic tripod

Legs that go down while the structure rises, numbers on the wire, and legs that follow a slope.

Assembling and the gunner seat

Two parts become one turret, why a freely spawned weapon arrives empty, and a door that is blocked on a hillside.

Dedicated-server pitfalls

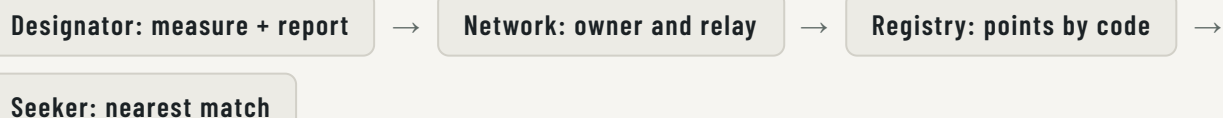
The short list: things that look fine in Workbench and fail on a server.

Integrating aircraft

What went wrong on helicopters: weapon racks, mode keys, phantom explosions, the missile camera, and testing.

The shape of it

Three kinds of participant that only know each other through one registry. A designator measures and reports a point with a four-digit code. The registry stores the point. A seeker asks for the nearest point with *its* code. Nobody holds a reference to anybody else, which is why a designator from this mod can guide a missile from a different one.



In the code: `GTG_SOFLAM_LaserDesignatorComponent` measures and reports, `GTG_SOFLAM_Relay` carries the message to the server when the player does not own the device, `GTG_SOFLAM_LaserTargetManager` is the registry, and `GTG_SOFLAM_LaserSeekerComponent` (or your own seeker that reads the registry) is the other end.

Rules that held up

RULE	WHY
Match by code, never by entity	The registry is the only coupling. Designator and seeker can come from different mods, different players, different vehicles.
Everything sent has an owner	A client that does not own the entity sends to the server and nothing arrives. No error, no log. See Who may send .
One writer per fact	Position is measured by one machine and broadcast. Every other machine derives what it needs from that.
Prefix every class	A name clash does not break one mod, it breaks the whole script module – for every player. The prefix is cheap insurance.
Measure, do not guess	Laser power limits, leg extension, offsets – all of them were run through by hand and confirmed on a server before they became numbers.

Make your mod compatible with the SOFLAM Core

The Core is a laser-target registry plus a SOFLAM and a tripod. Your mod can use it in three ways: read the registry (a guided weapon), set a code and fire (an aircraft or turret), or put a laser point into it (your own designator). This section says what to do in each case, in the order you will meet the problems.

IN PLAIN WORDS

Start here if you build a missile, an aircraft, a turret or a designator and want it to work with the SOFLAM Core: what you build, what the Core already does, and which names you may rely on.

HOW TO READ THE EXAMPLES

GTG_SOFLAM_* are names from the Core. MyMod_* in a snippet stands for *your* prefix – give every class of your mod one. Snippets are shortened excerpts. Log lines are quoted exactly as the code prints them (German), with a translation.

What do you want to build?

I WANT TO ...	WHAT YOU DO	PAGE
Make a missile, bomb or shell follow a SOFLAM laser	Give it a four-digit code and read the registry from its seeker (or attach the Core's seeker component).	2. Make a weapon follow the laser
Let the crew of an aircraft or vehicle set a code and fire guided weapons	A code holder on the turret, four dial actions in the cockpit, an injector that copies the code onto each missile, the right seat name.	3. Aircraft, vehicles and turrets
Mark targets from something that is not a SOFLAM (drone, turret)	Either put the Core's designator component on it, or report to the registry yourself through your own relay.	4. Report your own designator
Make my mod work with or without the Core	A small bridge addon that depends on both (a pattern; the published mods depend on the Core directly).	5. Optional integration

I WANT TO ...	WHAT YOU DO	PAGE
Find out why nothing happens on my server	Checklist, then symptom → cause → fix.	6. Testing and common errors

Whatever you build, do [step 1](#) first: the Core has to be a dependency of your addon.

What the Core does, and what you build yourself

The Core is the **meeting point**. It holds the laser points and brings them to every machine. It does not know your aircraft, your missile or your cockpit. Every aircraft mod that exists today built the parts around it itself:

THE CORE GIVES YOU	YOU BUILD
The registry of laser points, filled on every machine (<code>GTG_SOFLAM_LaserTargetManager</code>).	The seeker that reads it, in your missile (the AH-64 and AH-1Z mods each wrote their own).
A SOFLAM that publishes points: handheld, or assembled on a tripod and used as a gunner.	The code holder and the four code dials on your aircraft; the Core's dial action only works on a device that carries a designator.
Input actions and default keys for designating, ranging, tracking, vision modes and laser power.	The injector that copies the code onto a missile when it is fired.
The relay that gets messages to the server when the player does not own the device (inside the designator).	The HUD marker for laser spots, the seat check, flight profiles and cockpit screens.
Night/thermal vision, the IR gate and the visible beam as components you can add to any device with an optic.	For a designator that is not the Core's: your own relay (step 4 shows how).

The public surface

These are the names that foreign code in the existing mods really uses or that the Core documents as stable (the registry's methods are named a promise in the Core's design notes). Everything else in the

Core is internal and may change.

CLASS OR NAME	ROLE	WHAT FOREIGN CODE DOES WITH IT	SEEN IN
GTG_SOFLAM_LaserTargetManager (a GameSystem)	The registry	GetInstance() and check for null every time. Read: GetTargetsByCode , GetAllTargets , GetClosestTargetByCode , GetTarget , GetTargetCount . Publish: CreateTarget , UpdateTarget , RemoveTarget .	AH-64, AH-1Z
GTG_SOFLAM_LaserTarget	One registry entry	GetPosition() , GetCode() , GetCreatorId() , GetLetzteMeldung() (time of the last report).	same
GTG_SOFLAM_LaserSeekerComponent	Seeker you can attach to a missile or turret	SetLaserCode , SetSearchActive , IsLaserTargetFound , GetLaserTargetPosition , GetDirectionToLaserTarget , GetSeekerAngles , static FindSeekerById ; override the protected SelectTarget / AcceptTarget .	no checked mod attaches it

CLASS OR NAME	ROLE	WHAT FOREIGN CODE DOES WITH IT	SEEN IN
GTG_SOFLAM_LaserDesignatorComponent	The designator on a device	SetLaserActive , IsLaserActive , SetLaserCode , GetLaserCode , SetActionsEnabled , GetCreatorId , SetCameraEntity , GetLaserPower / SetLaserPower ; static FindOn(entity) . A subclass may override the protected ComputeTargetPosition .	the Core's own prefabs; its test emitters in the AH-64 / AH-1Z mods only publish through the registry
GTG_SOFLAM_SoflamDesignatorComponent	The designator class that sits on the SOFLAM prefabs (five-ray beam bundle)	Never search for it by its own class; use GTG_SOFLAM_LaserDesignatorComponent.FindOn(entity) , which finds both.	Core
GTG_SOFLAM_Relay and the GTG_SOFLAM_Send* methods on SCR_CharacterControllerComponent	The route to the server	Used by the designator. It resolves a RplId to a GTG_SOFLAM_LaserDesignatorComponent and nothing else, so a device without that component needs its own relay.	Core
GTG_SOFLAM_LaserCodeDigitAction	One code wheel (user action)	Put four on a device that carries a designator, with m_iDigitIndex 0..3. Digits run 1..8.	Core prefabs

CLASS OR NAME	ROLE	WHAT FOREIGN CODE DOES WITH IT	SEEN IN
GTG_SOFLAM_NVState	“Can this viewer see infrared?”	CanLocalPlayerSeeIR() ; set SetTemporaryNVState(true) while the player looks through your night-vision optic.	Core
GTG_SOFLAM_VisionModeComponent , enum GTG_SOFLAM_EVisionMode	Normal / night vision / thermal on any device with an optic	Add it to the device; call SetInputEnabled(true) when the player starts looking through, false when they stop.	Core prefabs
GTG_SOFLAM_BeamComponent , GTG_SOFLAM_LightEntity , prefab GTG_SOFLAM_IRLaserLight.et	The visible IR beam	Add the beam component next to a designator and set m_sBeamLightPrefab .	Core prefabs
GTG_SOFLAM_SoflamSightsComponent	The optic of the SOFLAM; knows when you look through it (IsAiming())	Inherit the SOFLAM prefab rather than rebuilding it.	Core
GTG_SOFLAM_IsAimingCondition	Condition for the control-hint display	Reference it from your own AvailableActions.conf .	AH-64
Prefabs PEQ1_SOFLAM.et {F2A55A212EBAF03F} , PEQ1_SOFLAM_Mounted.et {6A48B5E000005012}	The carried and the turret-mounted SOFLAM	Inherit them (test emitters in the AH-64 / AH-1Z mods) or list them in an arsenal catalogue.	AH-64, AH-1Z

CLASS OR NAME	ROLE	WHAT FOREIGN CODE DOES WITH IT	SEEN IN
Input actions GTG_SOFLAM_DesignatorAction (T), _RangeFindAction (Y), _TrackingAction (U), _VisionModeAction (J), _ToggleNvThermalAction (mouse:button2), _GainUpAction / _GainDownAction (LSHIFT + wheel), _LaserPowerUpAction / _LaserPowerDownAction (H + wheel)	Keys of the SOFLAM	Keep the attribute defaults of the designator and vision components and your device gets these keys. Do not define actions with these names yourself.	Core

BE AWARE

The Core ships GTG_SOFLAM_LaserSeekerComponent , but none of the Core's prefabs and none of the mods checked for this page attaches it – they all read the registry directly. Both ways are described on the page about [guided weapons](#); the registry route is the one that is proven in play.

Which existing mod to copy

MOD	PATTERN IT SHOWS	DEPENDS ON THE CORE
Sage's AH-1Z – SOFLAM Integration (Workshop)	Aircraft: code holder, four cockpit dials, injector, seeker, HUD markers. The cleanest aircraft template.	Yes (direct, in <code>addon.gproj</code>)
AH-64 SOFLAME Integration (Workshop)	The same, plus cockpit screens and flight profiles. More than you need for the Core part.	Yes

Five rules for foreign code

1. **List the Core as a dependency**, and only one copy of it ([step 1](#)).
2. **Ask for the registry every time** and check for null:
 GTG_SOFLAM_LaserTargetManager.GetInstance() returns null when no world is loaded. Do not keep it in a field.

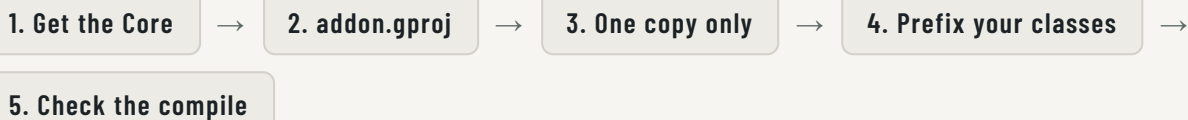
3. **Skip entries at the zero vector.** `CreateTarget` stores no position, so a point that was created but never reported sits at `0 0 0`.
4. **Prefix everything** you add, including RPC methods and input actions. A name clash takes down the whole script module for every player.
5. **Test on a real dedicated server with a separate client.** Most faults in this area are invisible in Workbench ([checklist](#)).

1. Add the Core as a dependency

One line in `addon.gproj`. Everything else in this section rests on it: the Core's classes exist for your scripts only if the Core is a dependency of your addon.

IN PLAIN WORDS

Put the Core's GUID into your `addon.gproj`. Without it, your scripts cannot see the Core's classes.



Step 1 – Get the Core

The Core is on the Arma Reforger Workshop: [SOFLAM Integration Core](#). Its own project file says everything about it that matters here:

EXAMPLE CODE - FREE TO USE

SOFLAM Integration Core / `addon.gproj`

```
GameProject {
  ID "SOFLAMIntegrationCore"
  GUID "6A48B52029EA0F9A"
  TITLE "SOFLAM Integration Core"
  Dependencies {
    "58D0FB3206B6F859"
  }
}
```

The only dependency of the Core is the game itself (`58D0FB3206B6F859`). It has no dependency on WCS Armaments or any other mod, so it does not tie you to a WCS version.

Step 2 – List its GUID in your `addon.gproj`

Add `6A48B52029EA0F9A` to the `Dependencies` of your addon. This is what the AH-1Z mod does (the other GUIDs are the game, the AH-1Z base mod and WCS Armaments):

EXAMPLE CODE - FREE TO USE

your addon / `addon.gproj`

```
GameProject {
  ID "MyMod"
  GUID "<your own GUID>"
  TITLE "My mod"
```

```
Dependencies {  
  "6A48B52029EA0F9A" "58D0FB3206B6F859" "68981B7716C2E9DB" "629B2BA37EFD577"  
}  
}
```

NOTE

Keep your own `ID` , `GUID` and `TITLE` ; only the dependency list is shared knowledge. A minimal mod needs just the game and the Core: `"58D0FB3206B6F859" "6A48B52029EA0F9A"` .

Step 3 – Use one copy of the Core only

List the GUID from step 2 and no other build of the Core. **Never load two copies of the Core at the same time** (for example a second build you test with): two addons that define the same class break the compile, for every player. Publish your mod against the GUID above.

Step 4 – Prefix your classes

Your classes, RPC methods and input actions need a prefix of your own. If two addons define a class with the same name, EnforceScript does not report “one mod is broken”; it breaks the **whole script module**, for every player on the server. The same goes for a compile error in any loaded mod. That is why the Core uses `GTG_SOFLAM_` for everything, and why the sibling aircraft mods use `GTG_AH64_` , `GTG_AH1Z_` and so on.

Step 5 – Check that it compiles

Open your addon in the Workbench and let the scripts compile. A reference to `GTG_SOFLAM_LaserTargetManager` in your script is resolved at compile time. If the Core is not a dependency, that line is a compile error, and the whole script module fails.

What happens when the Core is missing

SITUATION	RESULT
Your script mentions <code>GTG_SOFLAM_*</code> , the Core is not a dependency	Compile error in your addon. A compile error in any loaded mod takes the script module down for everybody on the server.

SITUATION

RESULT

The Core is a dependency, so your mod cannot do its job without it

Say **required** in your Workshop description, as the AH-64 mod does (“REQUIRED, please subscribe”).

You want the Core to be optional

A script reference cannot be optional. Use a bridge addon instead – [step 5](#).

NOTE

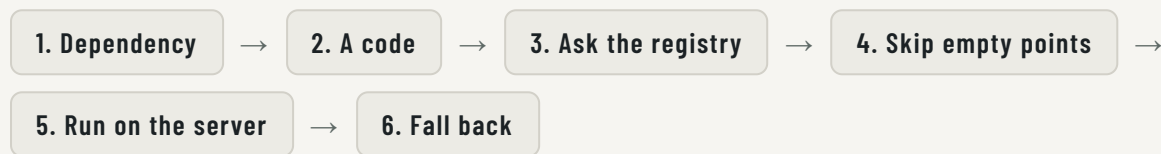
In the existing mods the dependency is a plain `addon.gproj` entry and every call into the Core is a normal compile-time reference. None of the mods checked reaches into the Core by name at runtime.

2. Make a weapon follow the laser

Your weapon does not need to know who is lasing. It needs a code and the nearest point with that code. There are two ways to get it; the first is the one every aircraft mod uses today.

IN PLAIN WORDS

A guided weapon needs two things: a four-digit code, and a seeker that asks the registry for the nearest laser point with that code.



Do [step 1](#) first. The principle is on the [registry](#) page: the match is by code only.

Route A – read the registry from your own seeker (proven)

The AH-64 and the AH-1Z both do this. The AH-64 and AH-1Z seekers extend the seeker of WCS Armaments (`WCS_Armament_LaserSeekerComponent`) and replace its target position. Your weapon's own seeker does the same job, whatever base class it has.

Step 2 – Give the weapon a code

The code is a string of four digits, each 1 to 8, for example `"1688"` (the default everywhere). The comparison is an exact string comparison, and an empty code never matches anything. Where the code comes from depends on the weapon:

- A launcher with a fixed code: an attribute on your launcher prefab with `"1688"` as the default, as the Core's own seeker has (`m_sDefaultLaserCode`).
- A weapon fired from an aircraft or turret: the code is copied onto the missile when it leaves the rail – see the injector on the page about [aircraft](#).

Step 3 – Ask the registry, skip empty points

This is the heart of every integration. It is the AH-1Z logic, cut to the point:

```
class MyMod_LaserSeeker : ScriptComponent
{
    protected string m_sTargetCode;

    void SetTargetCode(string code)
    {
        m_sTargetCode = code;
    }

    // true and the point, if somebody is lasing under our code
    bool FindLaserPoint(out vector point)
    {
        point = vector.Zero;
        if (m_sTargetCode.IsEmpty())
            return false;

        // ask EVERY time and check for null: no world loaded = no registry. Never keep it in a
        field.
        GTG_SOFLAM_LaserTargetManager manager = GTG_SOFLAM_LaserTargetManager.GetInstance();
        if (!manager)
            return false;

        array<GTG_SOFLAM_LaserTarget> targets = {};
        if (manager.GetTargetsByCode(m_sTargetCode, targets) == 0)
            return false;

        float bestDistSq = -1;
        foreach (GTG_SOFLAM_LaserTarget target : targets)
        {
            if (!target)
                continue;

            vector position = target.GetPosition();
            if (position == vector.Zero)
                continue; // created but never reported: CreateTarget stores no position

            float distSq = vector.DistanceSq(GetOwner().GetOrigin(), position);
            if (bestDistSq < 0 || distSq < bestDistSq)
            {
                bestDistSq = distSq;
                point = position;
            }
        }

        return bestDistSq >= 0;
    }
}
```

THE ZERO-VECTOR TRAP

`GetClosestTargetByCode(code, fromPosition)` exists, but it does **NOT**

skip entries at the zero vector. A point that was created and not yet reported would win if it is the nearest, and a missile would turn towards the world origin for a moment. The AH-64 and AH-1Z seekers loop over `GetTargetsByCode` instead.

Step 4 – Decide what “in view” means for your weapon

The registry knows nothing about your seeker's field of view. If you want one, apply it yourself. The AH-64 seeker compares the angle between the missile's forward axis and the direction to the point with `m_flaserAcquisitionFov`. Its ammo prefab sets that to `180`, on purpose: the stock default of 70 degrees made the hit rate depend on the direction of the shot relative to the target, because the target comes from a code match and not from real optics.

Step 5 – Run it where the missile flies, and tell the clients

A missile is simulated on the server. The AH-64 seeker returns at once on a proxy (`if (m_RplComponent && m_RplComponent.IsProxy()) return;`). That has a consequence: anything the seeker computes exists **only on the server**. The AH-64 mod showed a green “guided” marker on a client that never appeared on a dedicated server, because the flag was only set there. The fix was one attribute – a replicated property:

EXAMPLE CODE - FREE TO USE

GTG_AH64_LaserSeekerComponent.c (excerpt)

```
[RplProp()]
protected bool m_bGTG_ZielGefuehrt;    // "target is being guided"; without [RplProp] only the
server ever knows
```

Step 6 – Say what happens when there is no point

If the registry has no point for your code, do something sensible. The AH-64 seeker falls back to the behaviour of its base class (the gunner's sight direction) and carries on. Without a fallback, a missile fired without a laser simply falls out of the sky.

LOG THE REASON

The AH-64 seeker can print one line per quarter second and missile that says which of four states it is in:

OHNE CODE

(no code was handed over),

KEIN ZIEL ZUM CODE

(the registry has nothing under this code),

AUSSERHALB SICHTFELD

(point found, outside the field of view),

GEFUEHRT

(guided). The line starts with [GTG_AH64] [TELEMETRIE] and is off by default

(GTG_LOG_TELEMETRIE). Build the same into your own seeker; each state points at a different place.

Route B – attach the Core's seeker component

The Core has a ready-made seeker, GTG_SOFLAM_LaserSeekerComponent . It does the registry query, the distance limit and the angle calculation for you and replicates its on/off state and code. Treat this route as **less proven**: no prefab in the Core and none of the mods checked attaches it.

1. The entity needs a RplComponent ; without one the component cannot send anything.
2. Add the component. These are its attributes with their defaults:

EXAMPLE CODE - FREE TO USE

prefab text; values are the defaults

```
GTG_SOFLAM_LaserSeekerComponent "{<new GUID>}" {
  m_sSeekerId "AGM"
  m_sDefaultLaserCode "1688"
  m_fHorizontalAngleLimit 35
  m_fVerticalAngleLimit 35
  m_fRangeLimit 40000
  m_fCheckInterval 0.0333
}
```

m_sSeekerId tells several seekers on one entity apart (AGM , GBU , RKT); find one with GTG_SOFLAM_LaserSeekerComponent.FindSeekerById(entity, "AGM") .

m_SeekerOrigin (a point) moves the origin away from the entity origin.

3. **Switch it on.** The search loop only runs while the seeker is active; at spawn it is off. At launch call

EXAMPLE CODE - FREE TO USE

your launch code

```
GTG_SOFLAM_LaserSeekerComponent seeker =
GTG_SOFLAM_LaserSeekerComponent.FindSeekerById(missile, "AGM");
if (seeker)
{
  seeker.SetLaserCode(code);          // ignored if empty or unchanged
}
```

```
seeker.SetSearchActive(true);  
}
```

4. Read the result each frame: `IsLaserTargetFound()` first, then `GetLaserTargetPosition()`, or `GetDirectionToLaserTarget()` (world angles; the zero vector when the point is beyond the angle limits) or `GetSeekerAngles()` (angles relative to the entity's own heading, what a turret or a guidance law needs).
5. Know what “found” means: the seeker accepts a point if it is within `m_fRangeLimit`. The angle limits are **not** part of it; ask `IsTargetAnglesWithinConstraints()` yourself, and only after `IsLaserTargetFound()` is true.
6. To choose differently (nearest to the line of sight, moving targets first), derive from the class and override the protected `SelectTarget()`; override `AcceptTarget(target, distance)` to add conditions.

SAME TRAP

The component selects the nearest point with `GetClosestTargetByCode`, so it inherits the zero-vector trap above. If your missile could be fired in the first tenth of a second after a laser is switched on, override `SelectTarget()` and skip the zero vector.

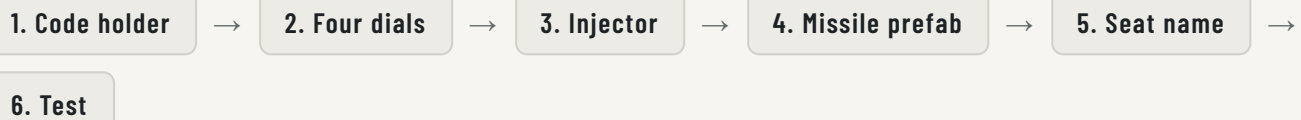
More on this component, the on/off replication and the origin point: [The seeker side](#).

3. Aircraft, vehicles and turrets

An aircraft is a *consumer*: the crew sets a code, fires a missile, and the missile asks the registry. The parts in between are yours. This page follows the AH-64 mod; the AH-1Z mod is a cleaned-up copy of it.

IN PLAIN WORDS

The crew of an aircraft needs a place that holds the code, buttons to change it, and something that copies it onto each missile. The Core does not ship these; the AH-64 and AH-1Z mods built them, and you can copy them.



Do [step 1](#) and read [how a weapon follows the laser](#) first.

WHAT IS THE CORE'S AND WHAT IS NOT

The names below belong to the AH-64 mod and start with `GTG_AH64_`. In your mod, use your own prefix. Projectile hooks such as `GetOnProjectileLaunched()` and the base classes `WCS_Armament_*` belong to WCS Armaments, which both aircraft mods use for the missile itself; they are not part of the Core. If your weapons are not WCS weapons, use whatever launch hook they have.

Step 1 – A code holder on the turret

The code is a property of the *aircraft*, not of the viewer: pilot and gunner must see the same one, a late joiner must get it, and the missile takes it along. So it lives in a component that replicates it. This is the AH-64's, shortened. It holds the code and nothing else:

EXAMPLE CODE – FREE TO USE

GTG_AH64_LaserCodeComponent.c (excerpt; logging left out)

```

class GTG_AH64_LaserCodeComponent : ScriptComponent
{
    [Attribute("1688", UIWidgets.Auto, "Laser code at start, four digits 1..8")]
    protected string m_sDefaultCode;

    protected string m_sCode;
    protected RplComponent m_RplComponent;

    override void OnPostInit(IEntity owner)
  
```

```

{
    super.OnPostInit(owner);
    m_sCode = m_sDefaultCode;
    m_RplComponent = RplComponent.Cast(owner.FindComponent(RplComponent));
}

string GetCode() { return m_sCode; }

void SetCode(string newCode)
{
    if (newCode == string.Empty || newCode == m_sCode)
        return;

    m_sCode = newCode;
    if (m_RplComponent)
        Rpc(GTG_RpcAsk_SetCode, m_sCode);           // client asks, the server distributes
}

[RplRpc(RplChannel.Reliable, RplRcver.Server)]
protected void GTG_RpcAsk_SetCode(string code)
{
    GTG_ApplySetCode(code);                          // the server's own copy
    Rpc(GTG_RpcDo_SetCode, code);                     // everybody else
}

[RplRpc(RplChannel.Reliable, RplRcver.Broadcast)]
protected void GTG_RpcDo_SetCode(string code)
{
    GTG_ApplySetCode(code);
}

protected void GTG_ApplySetCode(string code)         // shared effect; never sends
{
    if (code != string.Empty)
        m_sCode = code;
}

override bool RplSave(ScriptBitWriter writer)        // late joiners
{
    writer.WriteString(m_sCode);
    return true;
}

override bool RplLoad(ScriptBitReader reader)
{
    string code;
    if (!reader.ReadString(code))
        return false;

    if (code != string.Empty)
        m_sCode = code;

    return true;
}
}

```

Why it looks like this – the Ask/Do pair, the shared effect method that never sends, and `RplSave` / `RplLoad` – is explained on [Who may send](#); it is the same design as the Core's designator.

In the turret prefab the component needs no attributes. It looks for a `RplComponent` on its own entity; without one it only works locally and sends nothing:

EXAMPLE CODE - FREE TO USE

AH64D_Turret_X4_ATGM.et

```
GTG_AH64_LaserCodeComponent "{9E87DDFAC4FF0E35}" {  
}
```

Step 2 – Four dials in the cockpit

Four instances of one user action, one per digit, each counting 1 to 8 and then back to 1. This is the AH-64's version (shortened):

EXAMPLE CODE - FREE TO USE

GTG_AH64_CockpitLaserTuneAction.c (excerpt)

```
class GTG_AH64_CockpitLaserTuneAction : ScriptedUserAction  
{  
    [Attribute("0", UIWidgets.Auto, "Which digit of the code, 0..3")]  
    protected int m_iLaserCodeDigit;  
  
    protected GTG_AH64_LaserCodeComponent m_CodeComp;           // found lazily, from the vehicle  
    root downwards  
  
    override void PerformAction(IEntity pOwnerEntity, IEntity pUserEntity)  
    {  
        if (!GTG_FindeCodeHalter())  
            return;  
  
        string code = m_CodeComp.GetCode();  
        if (code.Length() != 4)                                // never count on a code that is not  
four characters  
            return;  
  
        array<string> digits = {};  
        for (int i = 0; i < 4; i++)  
            digits.Insert(code.Get(i));  
  
        int digit = digits.Get(m_iLaserCodeDigit).ToInt() + 1;  
        if (digit > 8)  
            digit = 1;  
  
        digits.Set(m_iLaserCodeDigit, digit.ToString());  
        m_CodeComp.SetCode(SCR_StringHelper.Join("", digits, true));  
    }  
  
    // LOCAL EFFECT ONLY, on purpose: see the pit below  
    override bool HasLocalEffectOnlyScript()  
    {  
        return true;  
    }  
}
```

A DIGIT THAT JUMPS BY TWO

The action returned `false` from `HasLocalEffectOnlyScript()` at first, so the engine replicated the *action* and ran `PerformAction` on the server and the clients, *and* the code holder sent the change as well. A relative step (“+1”) through two channels counts twice: the digit jumped by two. It does not show in a self-hosted game, because player and server are one machine. Let the holder be the only way the change travels.

The prefab text for the four dials (the same four lines for digit 0 to 3):

EXAMPLE CODE - FREE TO USE

AH64D_Turret_X4_ATGM.et (digits 0 and 1; 2 and 3 follow)

```
GTG_AH64_CockpitLaserTuneAction "{7C4E1A0000000140}" {
  ParentContextList {
    "Radio_Gunner"
  }
  "Sort Priority" 20
  m_iLaserCodeDigit 0
}
GTG_AH64_CockpitLaserTuneAction "{7C4E1A0000000141}" {
  ParentContextList {
    "Radio_Gunner"
  }
  "Sort Priority" 21
  m_iLaserCodeDigit 1
}
```

- `ParentContextList` names an **interaction context** that already exists on the vehicle (`Radio_Gunner` comes from the WCS AH-64D). It decides *where* the player stands to use it. Pick one that exists – the AH-64 mod chose the gunner's radio panel instead of inventing a position.
- The action and the holder are on different entities (cockpit and turret). The action therefore looks for the holder from the vehicle root downwards, and only when it is first needed, not in `Init`: at `Init` the instrument panel may not hang on the vehicle yet, and a failure there would be final.
- The AH-1Z mod needed `AllowCrossHierarchy 1` on the actions manager when action and holder are on different entities, and it puts one context point per digit (`GTG_LSS_Digit_1` to `_4`) on the vehicle.

Step 3 – An injector that copies the code onto each missile

The missile is a new entity every time. Something has to give it the code at the moment it leaves the rail. The AH-64 mod does it on the turret:

EXAMPLE CODE - FREE TO USE

GTG_AH64_LaserCodeInjectorComponent.c (excerpt)

```
class GTG_AH64_LaserCodeInjectorComponent : ScriptComponent
{
  protected GTG_AH64_LaserCodeComponent m_CodeHolder;
```

```

override protected void OnPostInit(IEntity owner)
{
    super.OnPostInit(owner);
    SetEventMask(owner, EntityEvent.POSTFRAME);    // look things up one tick later, not in
init
}

override event void EOnPostFrame(IEntity owner, float timeSlice)
{
    ClearEventMask(owner, EntityEvent.POSTFRAME);
    m_CodeHolder = GTG_AH64_LaserCodeComponent.FindOn(owner);
    // ... subscribe to the launch event of every weapon of the loadout:
    //     weapon.GetOnProjectileLaunched().Insert(OnProjectileLaunched);    (WCS)
}

void OnProjectileLaunched(Projectile projectile)
{
    if (!projectile || !m_CodeHolder)
        return;

    GTG_AH64_LaserSeekerComponent seeker = GTG_AH64_LaserSeekerComponent.Cast(
        projectile.FindComponent(GTG_AH64_LaserSeekerComponent));
    if (seeker)
        seeker.SetTargetCode(m_CodeHolder.GetCode());
}
}

```

The two ways a missile can leave without a code look identical from outside – it falls – so the injector reports them separately, as warnings: `[GTG_AH64][TELEMETRIE] Start OHNE Codehalter -- GTG_AH64_LaserCodeComponent am Turm nicht gefunden.` (“launch without code holder”: the component is missing from the turret prefab) and `[GTG_AH64][TELEMETRIE] Start OHNE Suchkopf -- GTG_AH64_LaserSeekerComponent nicht an der Rakete.` (“launch without seeker”: it is missing from the ammo prefab).

Step 4 – The missile prefab

The AH-64 mod switches the stock WCS laser seeker and flight computer off and puts its own on the ammo prefab:

EXAMPLE CODE - FREE TO USE

Ammo_Missile_AGM-114L_Laser.et (excerpt)

```

WCS_Armament_LaserSeekerComponent "{62A51932B80C9D78}" {
    Enabled 0
}
GTG_AH64_LaserSeekerComponent "{33BAD89EB72B67ED}" {
    m_fTargetCheckInterval 0.05
    m_fLaserAcquisitionFov 180
}
WCS_Armament_LaserGroundMissileFlightComputerComponent "{62B2C7B3F8499C97}" {

```

Enabled 0

```
}
```

The check interval matters: the prefab comment explains that 0.5 s let a missile at 150 - 250 m/s travel 75 - 125 m between two target updates, which limited the final accuracy hard. 0.05 s fixed that.

The turret and pod prefabs are **overrides**: they claim the same resource GUID as the WCS originals (through the `.et.meta` file), so the unchanged vehicle prefab, which already refers to turret and pod by GUID, now points to the mod's version. That is how the AH-64 mod got its code dials onto the existing Apache without a new vehicle variant.

Step 5 – Get the seat name right

Several things in an aircraft mod depend on *which seat* the player is in – HUD markers, the attack-mode listener. The name is not the same in every aircraft:

AIRCRAFT	WEAPON SEAT (COMPARTMENT NAME)	WHERE IT IS SET
AH-64D	gunner	<code>m_sPilotCompartmentUniqueName "gunner"</code> in the turret prefab; <code>IsInCompartment(..., "gunner")</code> in the marker component
AH-1Z	copilot	<code>CompartmentUniqueName "copilot"</code> in the turret prefab and in the marker component

SEAT IS NOT CONTEXT

The

COMPARTMENT NAME

(where the player sits) and the

INTERACTION CONTEXT

(where an action can be used) are two different things. In one helicopter port the dials hung on the context `copilot` and never worked; the context name comes from the shared turret base and is `gunner` whatever the seat is called. Read both from the vehicle prefab; do not assume.

Step 6 – Know who owns the vehicle

The code holder sends its Ask straight from the component on the vehicle. That worked on a dedicated server in the AH-64 mod (its changelog says it was tested there). An RPC to the server only arrives if the sender **owns** the entity (see [Who may send](#)). The Core's own design notes list as still to be checked

whether a turret gunner owns the vehicle. If a dial does nothing on a server and the log is silent, this is the first suspect; the cure is the relay pattern of the [next page](#) (the message goes through the player's character).

Optional – markers, screens, and far-away operators

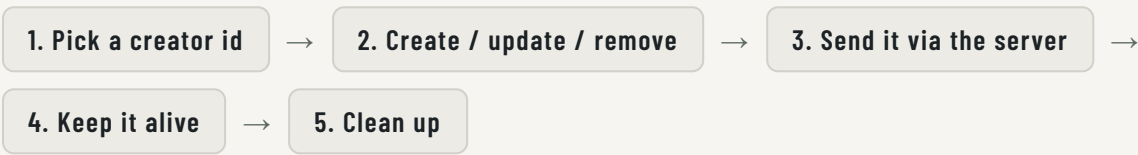
- **HUD marker.** The AH-64 and AH-1Z mods draw a diamond and the code per laser point (`GTG_AH64_LaserSpotIndicatorComponent`), reading the whole list with `GetAllTargets(array)`. Show it only while the player sits in the weapon seat.
- **Show the code where the crew looks.** A text overlay follows the head; the AH-64 mod moved the code onto a cockpit screen.
- **Operators far away.** Laser points of an operator more than about 1600 m from the gunner vanished, because the operator's character was not streamed that far. The AH-64 mod fixes it by streaming every character to every client: `RplComponent.EnableSpatialRelevancy(true)` plus `EnableStreamingConNode(identity, false)` per connected player, on the server, in a modded `SCR_ChimeraCharacter` and a modded `SCR_BaseGameMode.OnPlayerConnected` (class `GTG_SoflameCharacterRelevancy`, technique taken from a view-distance addon). The AH-1Z mod has the same hook; the Core itself does not contain it. If two of your mods carry it, load only one copy.

4. Report your own designator

Anything that can point at the ground can be a designator: it creates one registry entry, keeps its position fresh, and removes it. The hard part is not the three calls. It is getting them from the player's machine to everybody else's.

IN PLAIN WORDS

A drone, a turret or any other device that should mark a target puts its laser point into the registry. Either with the Core's designator component, or on its own through a relay on the player's character.



Do **step 1** first.

Which way: the Core's designator, or your own?

	CORE DESIGNATOR COMPONENT	YOUR OWN, CALLING THE REGISTRY
What it is	<code>GTG_SOFLAM_LaserDesignatorComponent</code> on your device: keys, rangefinder, tracking, laser power, relay, replication all included.	Three registry calls and a relay you write.
Fits	A device with an optic that a player looks through, like the SOFLAM.	Anything else: a drone camera, a turret.
Proven in	The SOFLAM prefabs of the Core.	The test emitters of the AH-64 and AH-1Z mods (without a relay).
Needs	A <code>Rp1Component</code> on the device; the target position comes from the camera (or, in the SOFLAM subclass, from the optic).	Any entity with a <code>Rp1Component</code> as creator id.

Way 1 – the Core's designator component

1. The device needs a `RplComponent` . Without it the component prints `[GTG_SOFLAM] Designator ohne RplComponent an der Traegerentitaet -- ohne die gibt es keine Ersteller-Id und keine Replikation.` once and stays quiet.
2. Add `GTG_SOFLAM_LaserDesignatorComponent` (or inherit a SOFLAM prefab). The defaults give you the keys T (designate), Y (range only) and U (track), code `"1688"` , 10 reports per second at most. The attributes are on the [designator](#) page.
3. Tell it when the player operates the device: `designator.SetActionsEnabled(true)` hooks the keys up, `SetActionsEnabled(false)` removes them (and switches the laser off, unless you pass `true` as the second argument). The SOFLAM's optic calls these when the player starts and stops looking through it.
4. Tell it where the camera is, if it is not the player's: `SetCameraEntity(entity)` . Without one, it measures from the active game camera.
5. For a different beam geometry, derive and override `ComputeTargetPosition()` – and ask `GTG_TrackingPoint()` first, as the Core's own two versions do, or tracking silently stops working in yours.

DERIVED CLASSES

The Core finds its designator with

`GTG_SOFLAM_LaserDesignatorComponent.FindOn(entity)` , which looks for the base class and then for `GTG_SOFLAM_SoflamDesignatorComponent` . A search for the base class alone does not find a derived class on the SOFLAM. Three symptoms – code wheel stuck, laser indicator stuck on “on”, designate key does nothing – had one cause, and none of them printed anything. If you derive your own, `FindOn` will not know it: call your own search, or make sure the code that needs it finds yours.

The Core's designator has no shipped example outside the SOFLAM prefabs. Test it on a dedicated server before you rely on it for a vehicle.

Way 2 – call the registry yourself

A designator that is not the SOFLAM reports into the registry with three calls on `GTG_SOFLAM_LaserTargetManager` . Everything else is networking.

Step 1 – Pick the creator id

The registry keys each entry by a `RplId` : the replication id of the entity that creates the laser point. One creator has exactly one entry; creating again replaces it. The Core's designator uses the id of its own device (`GetCreatorId()` returns `RplComponent.Id()`). A drone would use the drone's id. It must be valid (`creatorId.IsValid()`), otherwise `CreateTarget` and `UpdateTarget` do nothing. On an entity that exists in the world file from the start the id may not be valid yet in `EOnInit` – the AH-64 test emitter waits for the first `POSTFRAME` .

Step 2 – Three calls

EXAMPLE CODE – FREE TO USE

the three calls the Core's designator makes too

```
GTG_SOFLAM_LaserTargetManager manager = GTG_SOFLAM_LaserTargetManager.GetInstance();
if (!manager)
    return;                                     // no world: nothing to do, ask again next time

manager.CreateTarget(creatorId, code); // the entry exists, but with NO position yet
manager.UpdateTarget(creatorId, position, code); // position and code; creates the entry if
it is missing
manager.RemoveTarget(creatorId);
```

Call `UpdateTarget` straight away; it creates the entry when there is none (an update can even reach a late-joining client before the create). A point that was only created sits at the zero vector and every reader has to skip it. Mind the order of the arguments: `UpdateTarget(creatorId, position, code)` , not code first.

Step 3 – Send it through the server, from a place the player owns

The registry replicates nothing. Every machine fills its own copy, from a broadcast. So the three calls must run on **every** machine, and the way there is: the player's machine asks the server, the server applies it itself and broadcasts. And a client may only call a server RPC on something it owns. The Core's relay on `SCR_CharacterControllerComponent` is the player's own character – but it only resolves the id to a `GTG_SOFLAM_LaserDesignatorComponent` , so a drone cannot use it. You need a second one with your own names:

EXAMPLE CODE – FREE TO USE

pattern (Create and Remove work the same way)

```
modded class SCR_CharacterControllerComponent
{
    // the server: apply here, then tell everybody else
    void MyMod_ServerUpdate(RplId creator, string code, vector position)
```

```

{
    MyMod_ApplyUpdate(creator, code, position);
    Rpc(MyMod_DoUpdate, creator, code, position);
}

protected void MyMod_ApplyUpdate(RplId creator, string code, vector position)
{
    GTG_SOFLAM_LaserTargetManager manager = GTG_SOFLAM_LaserTargetManager.GetInstance();
    if (manager)
        manager.UpdateTarget(creator, position, code);
}

[RplRpc(RplChannel.Reliable, RplRcver.Server)]
void MyMod_AskUpdate(RplId creator, string code, vector position)
{
    MyMod_ServerUpdate(creator, code, position);
}

[RplRpc(RplChannel.Reliable, RplRcver.Broadcast)]
void MyMod_DoUpdate(RplId creator, string code, vector position)
{
    MyMod_ApplyUpdate(creator, code, position);
}

// Rpc() is protected: the call must stand INSIDE this class, so callers use this public
method
void MyMod_SendUpdate(RplId creator, string code, vector position)
{
    Rpc(MyMod_AskUpdate, creator, code, position);
}
}

```

And the sender, which the player's machine runs:

EXAMPLE CODE - FREE TO USE

the sender; Replication.IsServer() decides the route

```

static SCR_CharacterControllerComponent MyMod_Sender()
{
    PlayerController controller = GetGame().GetPlayerController();
    if (!controller)
        return null; // dedicated server: no local player

    IEntity character = controller.GetControlledEntity();
    if (!character)
        return null;

    return SCR_CharacterControllerComponent.Cast(
        character.FindComponent(SCR_CharacterControllerComponent));
}

// ...
SCR_CharacterControllerComponent relay = MyMod_Sender();
if (relay)
{
    if (Replication.IsServer())
        relay.MyMod_ServerUpdate(creatorId, code, position); // a broadcast does not deliver
to its own sender
    else

```

```
relay.MyMod_SendUpdate(creatorId, code, position);
```

```
}
```

- Look the sender up **every time**, from the controlled entity. The character changes at a respawn; a remembered reference points at a corpse the player no longer owns.
- All of them `Reliable`, including the position. A position still on its way could otherwise arrive *after* the remove and create the point again: a laser that never goes out. The Core's relay used an unreliable channel for the position for a while and had exactly this; entries stood for minutes, three at a time.
- Do not use the Core's `GTG_SOFLAM_SendeZielAktualisieren` and friends for a foreign device: their server side only knows `GTG_SOFLAM_LaserDesignatorComponent`. The call would arrive and do nothing.
- Give every method a prefix. You are adding to a vanilla class that every other mod can add to, too.

Step 4 – Keep it alive, but not too hard

The registry throws away any entry that has not been updated for **5 seconds** (checked once a second). So you must keep reporting, whether the point moves or not. The Core's designator reports:

- at most every `m_fUpdateInterval` = 0.1 s (ten per second),
- only if the point moved at least `m_fMinMoveDistance` = 0.5 m,
- but at least once every 2 seconds even if it did not move (a “life sign”). Without that, a steady laser would vanish after five seconds, just when it is needed.

Sending every frame also works, but on a reliable channel it is a lot of traffic. Take the Core's rhythm unless you have a reason.

When the operator can no longer see the target (looks away), keep sending the *last valid* point rather than stopping, so that clients who only come into range later still get a position.

Step 5 – Clean up

- Remove the entry when the laser goes off: `RemoveTarget(creatorId)`, through the same route.
- Remove it when the operator leaves. A first version often forgets this: a pilot who leaves a drone without switching off leaves a stale entry that survives until the world is reloaded, even a respawn. Remove it as soon as the device no longer belongs to the local player.
- Remove it when your entity is deleted (`OnDelete`). The Core's designator does this on every machine, not just the operator's.

- Do it **locally first, then send**. If the send fails, a removal that only travels over the network leaves the entry standing on every machine, including yours. The expiry is the net under this, not the plan.

5. Optional integration without a hard dependency

A script cannot say “use this class if it exists”: a reference to a class that is not there stops the compile. What does work is to keep the reference in an addon that only exists when both sides do.

IN PLAIN WORDS

If your mod should work with or without the Core, put the glue in a third addon that depends on both.

1. Hook in your mod → 2. Bridge addon → 3. Register → 4. Ask every time

What does not work

A reference to `GTG_SOFLAM_LaserTargetManager` in your main mod makes the Core a hard dependency, because the compiler resolves it. A runtime “if the Core is present” around it does not help – the line must compile first. A missing class breaks the script module, and `"ClassName".ToType()` can tell you whether a class exists but cannot call a method on it.

What works: a bridge addon

A PATTERN

This is a pattern, not something the Core ships. The published mods that use the Core (the AH-64 and the AH-1Z) simply depend on it. Use a bridge only if your main mod has to stay usable without the Core.

Your main mod does not depend on the Core. It has a small base class, a **provider**, that answers one question: where is the laser point for this code? A separate addon, the **bridge**, depends on your mod *and* the Core and fills that hook. If a player loads the bridge, your weapons follow SOFLAM lasers. If not, nothing changes.

1. **In your main mod**, define the question without knowing the Core: a base class with a static register/get and a few virtual methods that answer “no” by default. Your code asks the registered provider, and does something sensible when there is none.

```

class MyMod_LaserProvider
{
    protected static ref MyMod_LaserProvider s_Instance;

    static void Register(MyMod_LaserProvider provider)
    {
        s_Instance = provider;
    }

    static MyMod_LaserProvider Get()
    {
        return s_Instance;
    }

    bool IsAvailable()
    {
        return false;
    }

    bool FindPointByCode(string code, vector fromPosition, out vector point)
    {
        return false;
    }
}

```

2. In the bridge addon, list *both* in `addon.gproj` : your mod's GUID and the Core's `6A48B52029EA0F9A` . Derive the provider and answer from the registry:

```

class MyMod_SoflamLaserProvider : MyMod_LaserProvider
{
    override bool IsAvailable()
    {
        return GTG_SOFLAM_LaserTargetManager.GetInstance() != null; // ready once the world has a registry
    }

    override bool FindPointByCode(string code, vector fromPosition, out vector point)
    {
        point = vector.Zero;
        if (code.IsEmpty())
            return false;

        GTG_SOFLAM_LaserTargetManager manager =
        GTG_SOFLAM_LaserTargetManager.GetInstance();
        if (!manager)
            return false;

        array<GTG_SOFLAM_LaserTarget> targets = {};
        if (manager.GetTargetsByCode(code, targets) == 0)
            return false;

        bool found = false;
        float bestDistSq = 0;
    }
}

```

```

foreach (GTG_SOFLAM_LaserTarget target : targets)
{
    if (!target)
        continue;

    vector position = target.GetPosition();
    if (position == vector.Zero)
        continue;                                     // created, never reported

    float distSq = vector.DistanceSq(fromPosition, position);
    if (!found || distSq < bestDistSq)
    {
        found = true;
        bestDistSq = distSq;
        point = position;
    }
}
return found;
}
}

```

3. **Register it**, and do it twice: once when the game mode starts (`modded SCR_BaseGameMode.OnGameStart`), once when the first character is created (`modded SCR_ChimeraCharacter.EOnInit`). An empty Workbench test world has no game mode, so only the second path runs there. Make the registration safe to repeat: do nothing if a provider of your kind already stands.
4. **Ask every time**. The provider keeps no world state and fetches the registry for each call, so it survives a world change.

The other direction: your mod calls an optional mod by name

What *can* be done at runtime is calling a method on an object by its name. The Core does it to be friendly to RHS night-vision goggles without depending on RHS:

EXAMPLE CODE - FREE TO USE

GTG_SOFLAM_NVState.c

```

static bool CanLocalPlayerSeeIR()
{
    if (s_bTemporaryNV)
        return true;

    SCR_PlayerController playerController =
    SCR_PlayerController.Cast(GetGame().GetPlayerController());
    if (!playerController)
        return false;

    ScriptModule scriptModule = GetGame().GetScriptModule();
    if (!scriptModule)
        return false;

    bool nvOff = true;

```

```
    if (scriptModule.Call(playerController, "RHS_IsNV0ff", false, nv0ff))    // false if the
method does not exist
        return !nv0ff;

    return false;
}
```

If RHS is loaded, the call returns the real answer; if not, it fails and only the Core's own state counts. The vanilla scripts use the same call (the AI behaviour node `SCR_AIDecoIsEmpty` calls `IsEmpty` on an object by name). Limits:

- You need an **object** to call it on. For a static method or a class you cannot name, this gives you nothing.
- It returns whether the call worked; the result comes back through the last argument.
- The Core asks on every call and does not remember a failure. If you copy the pattern into something that runs every frame, remember the first failure and stop asking.
- The AH-64 mod looks classes up by name for RHS, too
(`"RHS_VehicleNightVisionComponent".ToType()`) to find out whether they exist.

NOT PROVEN

A way to reach `GTG_SOFLAM_LaserTargetManager` without a compile reference could not be shown from the code of the published mods; both use a dependency. If you find one, test it on a dedicated server first.

6. Testing and common errors

Most errors in this area are silent. They look the same from outside (nothing happens, or a missile falls) and have different causes. This page is the order in which to check, and the log lines that tell the causes apart.

IN PLAIN WORDS

A checklist for a dedicated server test, and a table of symptoms with the log lines the Core really prints.

The test

1. **Run a real dedicated server and a separate client.** In Workbench play client and server are one machine; ownership, relay and replication problems cannot show.
2. **Use two players** when you can: one on the SOFLAM, one in the aircraft, far enough apart (try more than 1600 m).
3. **Use a code nobody else uses**, and remove always-on test devices from the world. The nearest point with a matching code wins, whoever made it.
4. **Know the Core's net log.** `GTG_LOG_NETZ` is a constant in `GTG_SOFLAM_LaserDesignatorComponent.c`, off in the published version. Switched on, every send and every effect prints a line like `[GTG_SOFLAM][NETZ] SENDE Ziel anlegen | Rolle Proxy-ohne-Besitz | Code 1688 | Laser true`; a `SENDE` without a following `WIRKT` on the server is the signature of a dropped message. Build the same into your own components (and keep it off when you publish).
5. **Look for lines that begin with** `[GTG_SOFLAM]` in both logs, server and client (`script.log` / `console.log`). Switch your own diagnostics off before you publish; a switch left on fills the log.
6. **Test a late joiner:** connect after the laser is on, and check that code and point arrive.
7. **Test the switch-off:** laser off, leave the device, delete it. The point must disappear (at the latest after 5 seconds).

Symptom, cause, fix

SYMPTOM	CAUSE	FIX
The script module does not compile; your mod references <code>GTG_SOFLAM_*</code>	The Core is not a dependency of your addon, or two copies of the Core are loaded.	Step 1 . Use the GUID from step 2, and only one copy.
Laser on, nothing arrives; the client log says <code>[GTG_SOFLAM] Designator sendet ohne Besitz der Traegerentitaet UND ohne erreichbares Relais -- der Server verwirft diese Nachrichten (Rp1Docs: RPC invoke from the client, Not Owner -> Dropped). GTG_SOFLAM_Relay.Sender() liefert nichts: kein PlayerController oder kein gesteuerter Charakter.</code>	The client does not own the device and has no character to relay through (“sends without owning the carrier entity and without a reachable relay”). Respawn window, spectator, or an entity not controlled by a player.	Make sure the player controls a character. For your own device build the relay of step 4 .
The same, but the Core's seeker says <code>[GTG_SOFLAM] Suchkopf AGM sendet ohne Besitz der Traegerentitaet -- der Server verwirft diese Nachrichten. Siehe R1a in ENTWURFSVORGABEN.txt.</code> (AGM is your <code>m_sSeekerId</code>)	The Core's seeker component sends its on/off state and code to the server from an entity the client does not own, and has no relay.	Switch it on and set the code on the server (or the owner), or use your own seeker (route A).

SYMPTOM	CAUSE	FIX
The beam does not appear; [GTG_SOFLAM] Kein Strahl seit einer Sekunde: ... with one of five reasons	kein Designator an dieser Entitaet (the beam component has no designator next to it); Ersteller-Id ungueltig -- Traegerentitaet ohne RplComponent; Zielablage nicht gefunden (the registry system is not running; see below); kein Ziel unter dieser Ersteller-Id eingetragen (the message never arrived, ownership); Ziel eingetragen, aber ohne Position (created, never reported).	Each reason names its own place. Only reported when the same reason stands for a full second, so the normal start-up does not warn.
[GTG_SOFLAM] Strahlkomponente ohne Licht- Prefab -- es wird kein Strahl gezeichnet. Feld 'Prefab der Lichtquelle' am Prefab setzen.	m_sBeamLightPrefab is empty on the beam component.	Set it to GTG_SOFLAM_IRLas erLight.et .
GTG_SOFLAM_LaserTargetManage r.GetInstance() is always null	The system needs one update point <i>and</i> an entry in the systems config. The Core has both; check that the Core is really loaded.	See the registry page.
A missile falls out of the sky after launch	Four causes that look the same: no code handed over, no point under the code, point outside the seeker's field of view, or the flight computer. Also the upper row of a pod (see aircraft).	Log which of the four (step 6); the AH-64 prints OHNE CODE, KEIN ZIEL ZUM CODE, AUSSERHALB SICHTFELD, GEFUEHRT.
[GTG_AH64][TELEMETRIE] Start OHNE Codehalter ... / Start OHNE Suchkopf ...	The code holder component is missing from the turret prefab / the seeker component is missing from the ammo prefab.	Add them (step 1 and 4).
A missile turns towards the world origin for a moment	A point that was created and not yet reported sits at 0 0 0 .	Skip the zero vector in your seeker.

SYMPTOM	CAUSE	FIX
Missiles steer to a point where nobody is lasing	A stale entry. The expiry removes it after 5 seconds without a report; if it lasts longer, something keeps reporting.	Find who reports; remove the entry when the operator leaves.
A laser point disappears after five seconds although the operator holds still	A publisher that only reports when the point moves.	Re-report at least every 2 seconds (step 4).
Targets vanish for the gunner when the operator is more than about 1600 m away	The operator's character is not streamed that far.	Stream characters to all clients (see far-away operators).
A code dial jumps by two	The change travels twice: through action replication and through the holder.	<code>HasLocalEffectOnlyScript()</code> returns <code>true</code> .
Dials do nothing in the cockpit	Wrong interaction context, or the holder is not found (it hangs on another entity).	Compare compartment name and context name (step 5); search the holder from the vehicle root.
Attack mode keeps the first value on the server	A listener on an input action that was renamed (WCS 8.3.0).	Input actions get renamed.
Thermal filter stays on the camera, the screen is white	The filter was cleared on a different camera id than it was set on.	The camera-id trap.
Everything in the log is quiet and still nothing works	Ownership: the message is dropped silently.	Who may send.

NOTE

Measure before you change. In nearly every one of these cases the first explanation was wrong, and the second came from the log.

Target registry

The registry is the contract between everything that marks and everything that seeks. It stores one point per designator, matches by code, and forgets entries nobody removed.

IN PLAIN WORDS

One place that stores every active laser point. Anyone who searches asks it for points with a matching code.

HOW TO HOOK THIS UP IN YOUR OWN MOD

To use the registry from your own mod, see [Make a weapon follow the laser](#) (reading) and [Report your own designator](#) (writing).

Match by code, not by entity

A seeker does not know which device is lasing it. It knows a four-digit code and asks for the nearest point that carries it. That is what lets a handheld device, a tripod, a vehicle turret and an aircraft all designate for the same missile – and it is what makes a code a real thing the player sets rather than an internal link.

One entry is a `GTG_SOFLAM_LaserTarget`. It is deliberately small:

EXAMPLE CODE - FREE TO USE

GTG_SOFLAM_LaserTargetManager.c (excerpt)

```
class GTG_SOFLAM_LaserTarget
{
    protected RplId m_CreatorId;        // who reported it
    protected vector m_vPosition;
    protected string m_sCode;

    //! World time of the last report. The basis of the expiry, see below.
    protected float m_fLetzteMeldung;

    void GTG_SOFLAM_LaserTarget(RplId creatorId, string code) { m_CreatorId = creatorId; m_sCode
= code; }

    RplId GetCreatorId()    { return m_CreatorId; }
    vector GetPosition()    { return m_vPosition; }
    string GetCode()        { return m_sCode; }
    float GetLetzteMeldung() { return m_fLetzteMeldung; }
    // plus SetPosition(), SetCode() and MeldungGesehen(time) for the registry itself
}
```

NOTE

The creator id sits *inside* the entry. If it only lived as the map key, every consumer would need to know the map's structure to reach it, and the structure could never change again. A consumer only needs `GetAllTargets()` .

NAMES

Some member names are German: `GetLetzteMeldung` is “last report”. They are public; they are what you have to type.

A game system, not a singleton

A static singleton survives a world change. Someone then has to remember to empty it on every start, and if they forget, seekers of the new round home in on points from the old one. A `GameSystem` hangs on the world: the engine creates it on load and throws it away on unload, with nothing left over. The pattern is taken from a vanilla system (`SCR_ScenarioFrameworkCallQueueSystem`).

EXAMPLE CODE - FREE TO USE

GTG_SOFLAM_LaserTargetManager.c (excerpt)

```
class GTG_SOFLAM_LaserTargetManager : GameSystem
{
    protected ref map<RplId, ref GTG_SOFLAM_LaserTarget> m_mTargets = new map<RplId, ref
GTG_SOFLAM_LaserTarget>();

    override static void InitInfo(WorldSystemInfo outInfo)
    {
        outInfo
            .SetAbstract(false)
            .AddPoint(ESystemPoint.FixedFrame);    // without any update point the engine never
creates the system
    }

    // can be null when no world is loaded: EVERY caller must check
    static GTG_SOFLAM_LaserTargetManager GetInstance()
    {
        World world = GetGame().GetWorld();
        if (!world)
            return null;

        return
GTG_SOFLAM_LaserTargetManager.Cast(world.FindSystem(GTG_SOFLAM_LaserTargetManager));
    }
}
```

The second half is the systems config. It extends the *vanilla* file, so the file's `.meta` carries the vanilla GUID `{86E953538A28A98D}` for `Configs/Systems/ChimeraSystemsConfig.conf` ; with a GUID of its own the file would be packed and never read:

```
SystemSettings : "{45C53F06BA17238D}configs/Systems/SystemsConfig.conf" {
  Systems +{
    GTG_SOFLAM_LaserTargetManager "{6A48B5D000000201}" {
    }
  }
}
```

PITFALL

Two things have to be true before `FindSystem()` returns anything. The class needs at least one update point, even if it never uses it, and it has to be listed in the systems config. The first version of this registry was commented as “deliberately without an update point” – the engine then never created it, `FindSystem()` returned null, and the laser did nothing for days. The only symptom was the beam component's `Zielablage nicht gefunden` (“registry not found”).

Every machine has its own copy

The registry replicates nothing by itself. Every machine fills its own copy from the broadcast RPCs of the designators (see [Who may send](#)). That is deliberate: the entries are small and change often, and a replication component of their own would not pay off. The consequence for you: anything that writes into the registry has to run on **every** machine, not only on the server.

One target per designator, and update before create

A designator has exactly one target, so reporting again replaces the old one. The update path creates the entry when it does not exist yet. That is not tidiness: on a client that joins late, the first message can be an update that arrives before the create. A registry that drops it leaves the client blind until the next create.

```
void UpdateTarget(RplId creatorId, vector position, string code)
{
    if (!creatorId.IsValid())
        return;

    GTG_SOFLAM_LaserTarget target = m_mTargets.Get(creatorId);
    if (!target)
    {
        target = new GTG_SOFLAM_LaserTarget(creatorId, code);
        before the create
        m_mTargets.Set(creatorId, target);
    }
    // an update may reach us
```

```

target.SetPosition(position);
target.SetCode(code);
target.MeldungGesehen(GTG_Weltzeit());
}

```

`CreateTarget(creatorId, code)`, on the other hand, stores **no position**. An entry that was created and never updated sits at the zero vector. Readers must skip it ([how](#)); `GetClosestTargetByCode` does not.

Keep the map private

A public map is manipulated from outside, including by mods you have never seen. After that the data structure cannot change without breaking all of them. The map is `protected`; the public surface is a set of methods, and the Core's design notes call their names and signatures a promise:

METHOD	WHAT IT DOES
<code>static GetInstance()</code>	The registry of the current world, or null.
<code>CreateTarget(RplId creatorId, string code)</code>	New entry for a designator, replacing its old one. No position yet.
<code>UpdateTarget(RplId creatorId, vector position, string code)</code>	Position and code; creates the entry if missing; stamps the time.
<code>RemoveTarget(RplId creatorId)</code>	Removes the entry.
<code>GetTarget(RplId creatorId)</code>	The entry of one designator, or null.
<code>int GetAllTargets(notnull out array<GTG_SOFLAM_LaserTarget> outTargets)</code>	All entries; returns the count.
<code>int GetTargetsByCode(string code, notnull out array<GTG_SOFLAM_LaserTarget> outTargets)</code>	All entries with this code. An empty code returns 0.
<code>GetClosestTargetByCode(string code, vector fromPosition)</code>	The nearest entry with this code, or null. Selecting the nearest belongs here, otherwise every seeker writes it a little differently.

METHOD

WHAT IT DOES

```
int GetTargetCount() ,
```

Count; clear all.

```
ClearTargets()
```

Expiry: the last safety net

A target is created over the network and removed over the network. When the return path fails, it stays. In one server run three entries stood unchanged for minutes in an open field, their device long gone. A seeker takes the nearest match, so a corpse right in front of the helicopter beats the real target at 1600 m.

Every report stamps the entry. The registry's own update point sweeps once a second and removes everything that has not been reported for **five seconds** (`GTG_VERFALL_MS = 5000` , `GTG_VERFALL_TAKT_MS = 1000`). The individual leaks were fixed one by one; the expiry is the net under all of them.

EXAMPLE CODE - FREE TO USE

GTG_SOFLAM_LaserTargetManager.c (excerpt; the log line is left out)

```
protected static const float GTG_VERFALL_MS = 5000;           // five seconds without a report
protected static const float GTG_VERFALL_TAKT_MS = 1000;       // sweep once a second

override event protected void OnUpdatePoint(WorldUpdatePointArgs args)
{
    float jetzt = GTG_Weltzeit();
    if (jetzt < m_fNaechsterVerfallslauf)
        return;

    m_fNaechsterVerfallslauf = jetzt + GTG_VERFALL_TAKT_MS;

    // count BACKWARDS: removing shifts the following elements, going forwards would skip
    // entries
    for (int i = m_mTargets.Count() - 1; i >= 0; i--)
    {
        GTG_SOFLAM_LaserTarget target = m_mTargets.GetElement(i);
        if (!target)
        {
            m_mTargets.Remove(m_mTargets.GetKey(i));
            continue;
        }

        if (jetzt - target.GetLetzteMeldung() < GTG_VERFALL_MS)
            continue;

        m_mTargets.Remove(target.GetCreatorId());
    }
}
```

Why time, and not “does the entity still exist”: a check with `Replication.FindItem` would be wrong on a client. A remote device is often not streamed there; its id cannot be resolved although the device lives and lases. The time since the last report is the same question on every machine.

What this means for a publisher: report at least every few seconds, even when the point does not move. The Core's designator sends a “life sign” every 2 seconds for exactly this reason ([The designator](#)).

NOTE

Switching the laser off is the normal way an entry expires, so the log line for an expired target – `[GTG_SOFLAM][VERFALL] Ziel ... seit ... s ohne Meldung -- entfernt` – is off in regular operation (`GTG_LOG_VERFALL = false`).

The designator

One component on whatever emits the laser: a handheld device, a vehicle turret, an aircraft. It finds the point, reports it to the registry and keeps it current. What happens with the point afterwards is none of its business.

IN PLAIN WORDS

The part that aims the laser, measures the point and reports it. It marks a point, or follows a vehicle, or only measures the distance.

HOW TO HOOK THIS UP IN YOUR OWN MOD

To put the designator on your own device, or to report without it, see [Report your own designator](#).

Two modes, as on the real device

Designate measures and reports: seekers lock on. **Range only** measures and nothing else – no registry entry, no network message, no seeker. Someone who only wants to know how far something is does not give himself away and does not trigger anything.

Both keys are attributes, and an empty field switches that mode off on a prefab. A second device with different needs then edits data and never touches script. The component is

`GTG_SOFLAM_LaserDesignatorComponent`; the one on the SOFLAM prefabs is its subclass

`GTG_SOFLAM_SoflamDesignatorComponent`, which replaces only how the target point is found.

EXAMPLE CODE - FREE TO USE

GTG_SOFLAM_LaserDesignatorComponent.c (excerpt)

```
class GTG_SOFLAM_LaserDesignatorComponent : ScriptComponent
{
    [Attribute("GTG_SOFLAM_DesignatorAction", UIWidgets.Auto, "Name of the input action that
switches the laser")]
    protected string m_sToggleAction;

    [Attribute("true", UIWidgets.Auto, "true = toggle (once on, once off), false = hold")]
    protected bool m_bToggleMode;

    [Attribute("GTG_SOFLAM_RangefindAction", UIWidgets.Auto, "Own key: ONLY measure. Empty = no
rangefinder.")]
    protected string m_sRangefindAction;

    [Attribute("true", UIWidgets.Auto, "true = measures while held, false = one measurement per
press")]
    protected bool m_bRangefinderContinuous;
```

```
// ...  
}
```

The keys are hooked up and removed in **one** method with a bool, `SetActionsEnabled(bool enable, bool keepLaser = false)`. The optic calls it when the player starts and stops looking through the device:

EXAMPLE CODE - FREE TO USE GTG_SOFLAM_LaserDesignatorComponent.c (excerpt, empty-field checks left out)

```
void SetActionsEnabled(bool enable, bool keepLaser = false)
{
    if (enable == m_bActionsEnabled)
        return;

    InputManager inputManager = GetGame().GetInputManager();
    if (!inputManager)
        return;

    m_bActionsEnabled = enable;

    if (enable)
    {
        inputManager.AddActionListener(m_sToggleAction, EActionTrigger.DOWN, OnToggleAction);
        if (!m_bToggleMode)
            inputManager.AddActionListener(m_sToggleAction, EActionTrigger.UP, OnToggleAction);
        // ... rangefinder and tracking key the same way
    }
    else
    {
        inputManager.RemoveActionListener(m_sToggleAction, EActionTrigger.DOWN, OnToggleAction);
        if (!m_bToggleMode)
            inputManager.RemoveActionListener(m_sToggleAction, EActionTrigger.UP,
OnToggleAction);
        // ... the same lines mirrored

        // keepLaser: the device stands on a mount and keeps lasing while the player walks away.
        // Only the keys come off - otherwise the next sight would have two listeners per key.
        if (!keepLaser)
        {
            SetRangefinderActive(false);
            SetLaserActive(false);
        }
    }
}
```

PITFALL

Register and unregister in

ONE

method. A pair of separate methods lets a copy-paste turn the second into `AddActionListener` again, and the listeners then pile up on every cycle.

NOTE

Toggle is the right default for a designator. With hold you cannot hold the key, aim and dial the code at the same time, and a device you put down could never keep lasing.

The attributes

All of these can be set on the prefab. Defaults are the values in the code.

ATTRIBUTE	DEFAULT	MEANING
<code>m_sToggleAction</code>	<code>GTG_SOFLAM_DesignatorAction</code>	Input action that switches the laser. Empty = no designating.
<code>m_bToggleMode</code>	<code>true</code>	Toggle, or hold.
<code>m_sRangefindAction</code>	<code>GTG_SOFLAM_RangefindAction</code>	Input action that only measures. Empty = no rangefinder.
<code>m_bRangefinderContinuous</code>	<code>true</code>	Measures while held; false = one measurement per press, the value stays.
<code>m_sTrackAction</code>	<code>GTG_SOFLAM_TrackingAction</code>	Input action for tracking. Empty = no tracking.
<code>m_bTrackAimsHead</code>	<code>true</code>	A device on a mount pans after the tracked target.
<code>m_bTrackFeedback</code>	<code>true</code>	Show a message when tracking goes on or off.
<code>m_bTrackVehicleCentre</code>	<code>true</code>	Aim at the geometric centre of the vehicle, not at the hit point.
<code>m_fMaxDistance</code>	<code>4000</code>	Longest laser range in metres (the base class's single ray).
<code>m_sDefaultLaserCode</code>	<code>1688</code>	Laser code at start.
<code>m_fUpdateInterval</code>	<code>0.1</code>	Seconds between two position reports (0 = every frame).
<code>m_fMinMoveDistance</code>	<code>0.5</code>	A new report only after the point moved this many metres.

ATTRIBUTE	DEFAULT	MEANING
<code>m_sOverlayLayout</code> , <code>m_sOverlayWidgetName</code>	empty, <code>LaserON</code>	A layout shown while the laser runs, and the widget in it that is shown and hidden.
<code>m_bDrawDebugTarget</code>	<code>false</code>	Draw the target point as a red sphere.
<code>m_fLaserPowerMin</code> / <code>Max</code> / <code>Step</code>	<code>8</code> / <code>40</code> / <code>1</code>	Laser power (LV) limits and step per wheel notch.
<code>m_fSoflamRange</code> (SOFLAM subclass only)	<code>3500</code>	Longest measuring range in metres.

The code dials

The code is four digits, each 1 to 8 (`"1688"` by default). On the SOFLAM it is set with four copies of one user action, `GTG_SOFLAM_LaserCodeDigitAction` , which differ only in an attribute (`m_iDigitIndex` 0 to 3, start values 1, 6, 8, 8). A device with a different number of wheels gets another copy (and a default code of that length), and no script.

The action does not set the code on the device it sits on. It goes through `designator.SetLaserCode(newCode)` , because that is the one place that distributes a change to the server and everybody else. A wheel that only changed a local field would let the operator turn it while the seeker kept looking for the old code, without any message. The action runs *locally only* (`HasLocalEffectOnlyScript` is `true`), otherwise the engine would replicate the action *and* the designator would send the change: the digit jumped by two.

A point or a vehicle

Without tracking, the designator marks a *point*: every tick a fresh trace to wherever the device points. When the vehicle drives off, the mark stays on the ground.

With tracking (key U, `GTG_SOFLAM_TrackingAction`), remember *which entity* was hit and *where on it* – in the entity's own coordinate system. The world point is recomputed from the entity's current transform every time, so the mark travels with the sheet metal instead of with the line of sight.

EXAMPLE CODE - FREE TO USE

GTG_SOFLAM_LaserDesignatorComponent.c (excerpt)

```
// at lock-on (OnTrackAction), shortened
vector mins, maxs;
ziel.GetBounds(mins, maxs);
vector mitte = (mins + maxs) * 0.5;
```

```

// the hit point is the worse choice: it is wherever you happened to hold, say the edge of the
// roof,
// and it swings wildly when the vehicle turns. The centre of the bounds is stable.
if (vector.Distance(mins, maxs) > 0.5)
    m_vTrackLocal = mitte;

// every tick
protected vector GTG_TrackedWorldPoint()
{
    if (!m_TrackedEntity)
        return vector.Zero;

    vector mat[4];
    m_TrackedEntity.GetWorldTransform(mat);

    return mat[3]
        + mat[0] * m_vTrackLocal[0]
        + mat[1] * m_vTrackLocal[1]
        + mat[2] * m_vTrackLocal[2];
}

```

- **Only on a tripod, only vehicles, only while lasing.** Tracking needs a ball head to pan, a `Vehicle` somewhere up the parent chain of the entity that was hit, and a running laser. Otherwise a message says why (“Verfolgung nur auf dem Dreibein”, “Kein Fahrzeug unter dem Kreuz”, “Erst markieren, dann verfolgen”).
- **A trace of its own for the lock.** The laser's trace uses `TraceFlags.VISIBILITY` so that walls and hills block it. That flag also stops `TraceEnt` from being filled: visibility geometry counts as world, not as an entity. Measured on an armoured Humvee: the beam ended exactly on the vehicle and could not say on what. So the lock-on does a second trace without that flag.
- **The tracking state is replicated.** It used to be a local field. The server ran its own copy that was never tracking, fell back to the camera, and sent that position as authoritatively as the operator sent the correct one. A missile flew correctly for seconds and then turned away when one of these server messages arrived last. Now tracking goes through the same Ask/Do route as the target (`GTG_RpcAsk_SetTracking` / `GTG_RpcDo_SetTracking`, with the `RplId` of the tracked vehicle and the local point).
- **If you derive your own designator** and override `ComputeTargetPosition()`, ask `GTG_TrackingPoint()` first, as the base class and `GTG_SOFLAM_SoflamDesignatorComponent` both do. The tracking code first lived inside the base version and was dead code, because the prefabs use the subclass, which overrides without calling `super`.

Report rarely, but never stop

An early version sent a reliable packet every frame for every active designator. Now the report is throttled to the interval above and sent only when the point really moved – with one exception, a life sign

every two seconds, so that a steady laser does not expire in the registry after five seconds. Calculating is separate from sending: while the target moves the point is recomputed every frame (and the mount follows smoothly); only the sending is throttled, so network traffic does not change.

EXAMPLE CODE - FREE TO USE

GTG_SOFLAM_LaserDesignatorComponent.c (excerpt)

```
// from EOnFrame, shortened
float jetztLz = GetGame().GetWorld().GetWorldTime();
bool lebenszeichen = jetztLz - m_fLetztesLebenszeichen >= 2000;    // life sign every two
seconds

if (!lebenszeichen && m_bHasSentPos && vector.DistanceSq(targetPos, m_vLastSentPos)
    < m_fMinMoveDistance * m_fMinMoveDistance)
    return;    // standing still: no
traffic

m_fLetztesLebenszeichen = jetztLz;
m_vLastSentPos = targetPos;
m_bHasSentPos = true;

GTG_SendeZielAktualisieren(targetPos);
```

All messages are **reliable**. The unreliable channel has no ordering relative to the reliable one: a position still in flight could arrive after the switch-off and create the point again – a laser that never goes out. The path of a message is described on [Who may send](#).

Limits you measured, not guessed

The laser power is adjustable between two limits (8 and 40 LV, one step per wheel notch with H held). Both were found by driving the device through its range: at the bottom the beam is clearly weak but still there (at zero the player could not tell a broken device from a switched-off laser); at the top the bloom becomes noticeable from about 18 m, and beyond 40 the dot washes out half the picture. The default is not in the script but at the light prefab (`GTG_SOFLAM_IRLaserLight.et` , `LV 25`): until someone turns the wheel, the designator's value is -1, “nothing set”, and the prefab value applies. One truth in one place. Once set, the value replicates to everybody, late joiners included, because brightness belongs to the device and not to the viewer.

NOTE

Whoever changes a limit like this should run it through again. The numbers on these pages are the results of that, not of reasoning.

Who may send

The bug that took the longest: with a dedicated server, no laser and no aiming worked, and not a single error appeared. The server simply never heard the client.

IN PLAIN WORDS

On a server a client may only send to the server for things it owns. Otherwise the message vanishes without a trace. A relay on the player's character gets around that.

HOW TO HOOK THIS UP IN YOUR OWN MOD

If you write a device of your own, you need this page: [Report your own designator](#) uses the pattern. If a dial or a key does nothing on a server, start with the [checklist](#).

The rule

An Rpc addressed to the server is delivered only if the sending client **owns** the entity the component sits on. If it does not, the message is dropped – no error, no log line. The engine's routing table says so (`Rp1Docs.c`, “RPC invoke from the client”: Not Owner → Dropped). In single-player nothing shows, because client and server are one machine and the question of ownership is never asked.

The log of the first real server test looked like this on the client, from the moment the laser was switched on (shortened from the comment in `GTG_SOFLAM_Relay.c`; the Core prints its messages in German, translation below):

EXAMPLE CODE - FREE TO USE

```
(W) Designator sendet ohne Besitz der Traegerentitaet -- der Server verwirft diese Nachrichten
[NETZ] SENDE Ziel anlegen          | Rolle Proxy-ohne-Besitz | Laser true
[NETZ] SENDE Ziel aktualisieren    | Rolle Proxy-ohne-Besitz | Laser true
(W) Kein Strahl seit einer Sekunde: kein Ziel unter dieser Ersteller-Id
[NETZ] SENDE Ziel aktualisieren (weiter im Sekundentakt)
(not a single "WIRKT" line - "takes effect" - after switch-on)
```

Translation: “Designator sends without owning the carrier entity – the server drops these messages”; “SENDE” is send, “Rolle Proxy-ohne-Besitz” is role proxy without owner; “no beam for one second: no target under this creator id”. The proof that the way *back* was fine stood three minutes earlier in the same log: a “WIRKT Ziel entfernen” arrived with no “SENDE” before it – so it came over the network. Only the direction client → server failed.

PITFALL

Holding the device in the hand does not give the client ownership of the item entity. The role stayed “proxy without owner” while the device lay in the inventory. An early comment in the code asserted the opposite – and was the reason neither laser nor ball head reacted on the server.

A relay on the character

You need an entity the *player* owns. The character is always his. Vanilla already hangs server-bound methods on the character controller component (`SCR_CharacterControllerComponent` carries four `RplRcver.Server` methods), so the route is not just allowed, it is the one in use. The `SCR_PlayerController` itself carries no RPC of its own in vanilla; reaching it would need a prefab override, which is a load-order lottery with every other mod that does the same.

So the Core adds eight server-bound methods to the character controller (`GTG_SOFLAM_Relais*`) and a public method for each that sends it (`GTG_SOFLAM_Sende*`). Every pair is the same shape. The designator picks the route per message:

EXAMPLE CODE - FREE TO USE `GTG_SOFLAM_Relay.c` and `GTG_SOFLAM_LaserDesignatorComponent.c` (excerpts)

```
modded class SCR_CharacterControllerComponent
{
    // runs on the SERVER, with an id that came from a client - so with every check that belongs
    to that
    [RplRpc(RplChannel.Reliable, RplRcver.Server)]
    void GTG_SOFLAM_RelaisZielAktualisieren(RplId geraet, string code, vector position)
    {
        GTG_SOFLAM_LaserDesignatorComponent designator = GTG_SOFLAM_Relay.FindeGeraet(geraet);
        if (designator)
            designator.GTG_ServerZielAktualisieren(code, position);
    }

    // Rpc() is protected: the call must stand inside THIS class, so the designator calls this
    public method
    void GTG_SOFLAM_SendeZielAktualisieren(RplId geraet, string code, vector position)
    {
        Rpc(GTG_SOFLAM_RelaisZielAktualisieren, geraet, code, position);
    }
}

// in the designator
protected bool GTG_BrauchtRelais()
{
    if (!m_RplComponent)
        return false;

    if (Replication.IsServer())
        return false; // the server sends directly

    return !m_RplComponent.IsOwner(); // an owner sends directly, anybody else needs the relay
}
```

```

SCR_CharacterControllerComponent relais;
if (GTG_BrauchtRelais())
    relais = GTG_SOFLAM_Relay.Sender();

if (relais)
    relais.GTG_SOFLAM_SendeZielAktualisieren(GetCreatorId(), m_sLaserCode, targetPos);
else
    Rpc(GTG_RpcAsk_UpdateTarget, m_sLaserCode, targetPos);

```

If the relay is not available, the message goes the direct way anyway. It is lost then, but the behaviour is the same as before and the one-time warning below names the reason. A silent special route would be the worse of two bad options.

RELAY METHOD (SERVER-BOUND)	CHANNEL	CARRIES
GTG_SOFLAM_RelaisZielAnlegen	Reliable	create the target (device id, code)
GTG_SOFLAM_RelaisZielAktualisieren	Reliable	position and code. Unreliable until 2026-09-17, then changed: entries stood for minutes as corpses.
GTG_SOFLAM_RelaisZielEntfernen	Reliable	remove the target
GTG_SOFLAM_RelaisCode, GTG_SOFLAM_RelaisStaerke	Reliable	laser code, laser power
GTG_SOFLAM_RelaisZielen	Unreliable	aim angles of the ball head. A lost angle heals itself on the next tick; reliability would only build a queue.
GTG_SOFLAM_RelaisZurueck	Reliable	put the device back on the mount
GTG_SOFLAM_RelaisVerfolgung	Reliable	tracking on/off with target id and local point. A lost on/off flip stays wrong; there is no next tick that heals it.

Every send looks the sender up again from the controlled entity (`GTG_SOFLAM_Relay.Sender()`), never from a remembered reference: the character changes at a respawn, and an old reference points at a corpse the player no longer owns. On the server, `FindeGeraet` turns the device's `RplId` into a designator – with `GTG_SOFLAM_LaserDesignatorComponent.FindOn`, because a search for the base class does not find the subclass on the SOFLAM.

When there is neither ownership nor a relay, the designator says so once, so that the laser does not just silently not work:

EXAMPLE CODE - FREE TO USE

```
[GTG_SOFLAM] Designator sendet ohne Besitz der Traegerentitaet UND ohne erreichbares Relais --
der Server verwirft diese Nachrichten
(RplDocs: RPC invoke from the client, Not Owner -> Dropped). GTG_SOFLAM_Relay.Sender() liefert
nichts: kein PlayerController oder kein gesteuerter Charakter.
```

NOTE

Add only. The relay is a class that adds methods; it overrides nothing and calls nothing of vanilla's. And every name carries the prefix. A second mod that adds methods to the same vanilla class is fine as long as the names differ.

RPCs live in your own component

An earlier approach put all four RPC blocks into `modded class SCR_ChimeraCharacter`. Everybody who loaded the mod inherited that extension unasked. A component becomes a replication item simply by having `[RplRpc]` methods. The second gain: the broadcast lands on the *same* component on every machine, so the creator id does not have to travel with every packet – each machine derives it from its own carrier entity (`GetCreatorId()` is the id of the replication component). Only the relay, which sits on the character, has to carry the device's id.

Ask, Do, Apply: the receive path must not send

Every state change has three methods: **Ask** (client to server), **Do** (broadcast) and **Apply** (the effect). The server applies a change itself and then broadcasts (a broadcast does not deliver to its own sender). If the method that applies a received state also reported that state, a client that receives a broadcast would immediately send it back to the server, and the loop closes. A commented-out line used to hide exactly that. The public path reports, the receive path applies and stays silent.

EXAMPLE CODE - FREE TO USE

GTG_SOFLAM_LaserDesignatorComponent.c (excerpt)

```
[RplRpc(RplChannel.Reliable, RplRcver.Server)]
protected void GTG_RpcAsk_UpdateTarget(string code, vector position)
{
    GTG_ServerZielAktualisieren(code, position);
}

void GTG_ServerZielAktualisieren(string code, vector position)
{
    GTG_ApplyUpdateTarget(code, position);           // the server's own copy
    Rpc(GTG_RpcDo_UpdateTarget, code, position);    // everybody else
}

[RplRpc(RplChannel.Reliable, RplRcver.Broadcast)]
protected void GTG_RpcDo_UpdateTarget(string code, vector position)
```

```
{  
    GTG_ApplyUpdateTarget(code, position);           // never sends  
}
```

The server-side entry points (`GTG_ServerZielAnlegen` , `GTG_ServerZielAktualisieren` , `GTG_ServerZielEntfernen` , `GTG_ServerCode` , `GTG_ServerStaerke`) are public on purpose: the relay calls them from the outside. The seeker uses the same split (`GTG_RpcAsk_SetState` / `GTG_RpcDo_SetState` / `GTG_ApplySearchState`).

PITFALL

The Ask method must not start with “if this is the server, return”. A state change on the server would then never reach anybody else. It was a bug in the seeker of the code the Core replaced: no client ever learned whether a seeker was on or off.

Late joiners

State that arrives only as messages is gone for everybody who connects later. The designator implements `RplSave` and `RplLoad` for the state that matters – laser on or off, code, last sent position, laser power – or a late client sees nothing until the next report. On loading, a running laser re-enters the registry through the same `GTG_ApplyUpdateTarget` .

When you can see the problem

The Core has a switch for exactly this, `GTG_LOG_NETZ` (off in the published version). It prints one line per send and per effect with the role of the machine: `[GTG_SOFLAM][NETZ] SENDE <message> | Rolle <Server|Besitzer|Proxy-ohne-Besitz> | Code ... | Laser ...` and `... WIRKT <message> | ...` . A SENDE that is never followed by a WIRKT on the server was dropped; a WIRKT without a SENDE on the same machine came over the network. Put a tag on every line of your own mod and write the *role* into every network line.

The seeker side

A component on a missile, a turret or a fire-control computer. It does not care who is lasing or whether it is its own laser. It asks the registry and works out direction and angle.

IN PLAIN WORDS

The seeker is the other half: it asks the registry for the nearest point with its code, checks the range, and hands back a direction.

HOW TO HOOK THIS UP IN YOUR OWN MOD

Two ways to use a seeker in your own mod – read the registry yourself, or attach this component – are in [Make a weapon follow the laser](#).

Ask by code, then range

The seeker, `GTG_SOFLAM_LaserSeekerComponent`, takes the nearest registry point with its code and accepts it if it is within its range limit. Both decisions are protected methods you can override, so that nobody has to copy the component to choose differently:

EXAMPLE CODE - FREE TO USE

GTG_SOFLAM_LaserSeekerComponent.c (excerpt)

```

//! Which target does this seeker pick? Override this to choose differently - nearest to the line
//! of sight instead of to the origin, moving targets first.
protected GTG_SOFLAM_LaserTarget SelectTarget()
{
    GTG_SOFLAM_LaserTargetManager manager = GTG_SOFLAM_LaserTargetManager.GetInstance();
    if (!manager)
        return null;

    return manager.GetClosestTargetByCode(m_sLaserCode, m_vSeekerStart);
}

//! Does the seeker take this target? Default: the range limit only.
//! The ANGLE limits stay out on purpose: a target counts as found even if it is just outside the cone.
protected bool AcceptTarget(GTG_SOFLAM_LaserTarget target, float distance)
{
    return m_fRangeLimit <= 0 || distance <= m_fRangeLimit;
}

```

Whether the target can be flown to is a separate question, asked with

`IsTargetAnglesWithinConstraints()`, or answered by `GetDirectionToLaserTarget(true)` returning the zero vector outside the limits:

EXAMPLE CODE - FREE TO USE

GTG_SOFLAM_LaserSeekerComponent.c (excerpt)

```
vector GetDirectionToLaserTarget(bool checkConstraints = true)
{
    if (!m_bTargetFound)
        return vector.Zero;

    vector angles = GTG_AnglesTo(m_vTargetPosition);

    if (checkConstraints)
    {
        vector seekerAngles = GTG_ToSeekerAngles(angles);
        if (Math.AbsFloat(seekerAngles[0]) > m_fHorizontalAngleLimit
            || Math.AbsFloat(seekerAngles[1]) > m_fVerticalAngleLimit)
            return vector.Zero;    // found, but outside the cone
    }

    return angles;
}
```

ATTRIBUTE	DEFAULT	MEANING
m_sSeekerId	AGM	Name of this seeker; several on one entity are told apart by it.
m_sDefaultLaserCode	1688	Code at start.
m_fHorizontalAngleLimit / m_fVerticalAngleLimit	35 / 35	Angle limits in degrees, relative to the entity's heading.
m_fRangeLimit	40000	Largest distance in metres at which a target is still accepted (0 or less = unlimited).
m_fCheckInterval	0.033 3	Seconds between two checks (0.0333 = 30 Hz).
m_SeekerOrigin	empty	Origin point of the seeker. Empty = the carrier entity's origin.
m_bDrawDebug	false	Draw line of sight and direction.

The public methods: `SetSearchActive(bool)` (with the shorthands `ActivateSearch()` and `DeactivateSearch()`), `IsSeekerActive()`, `IsLaserTargetFound()`, `GetCode()` / `SetLaserCode(string)`, `GetLaserSeekerID()`, `GetLaserTargetPosition()`,

`GetLaserSeekerOrigin()` , `GetDistanceToTarget()` , `GetDirectionToLaserTarget(bool)` (world angles), `GetSeekerAngles()` (angles relative to the carrier's heading, what a turret or a guidance law needs) and the static `FindSeekerById(entity, id)` .

OFF BY DEFAULT

The seeker

DOES NOT SEARCH UNTIL IT IS SWITCHED ON

: `SetSearchActive(true)` sets the frame event. A freshly spawned seeker is off. State and code are replicated (Ask to the server, Do to everybody, `RplSave` / `RplLoad` for late joiners), and it warns once if it has to send without owning its entity: `[GTG_SOFLAM] Suchkopf AGM sendet ohne Besitz der Traegerentitaet -- der Server verwirft diese Nachrichten.` Siehe `R1a` in `ENTWURFSVORGABEN.txt`.

ZERO-POSITION ENTRIES

`GetClosestTargetByCode` does not skip entries at the zero vector (created, never reported), and neither does this component by default. Override `SelectTarget()` if that matters for your weapon.

NOTE

The Core's seeker does not log why it has no target. The AH-64 mod's own seeker does, and tells four states apart, each pointing at a different place:

OHNE CODE

(the carrier never handed one over),

KEIN ZIEL ZUM CODE

(nobody is lasing, or under another code),

AUSSERHALB SICHTFELD

(the alignment at launch is wrong, not the target),

GEFUEHRT

(guided). A missile that simply falls out of the sky is one of the four. Build the same into yours.

Several seekers on one entity

A carrier can have more than one: give each an id (`AGM` , `GBU` , `RKT`) and look it up by that id with `FindSeekerById` . Mount, weapon and seeker then stay separate things, and a rocket pod and a missile on the same turret each keep their own code.

A range limit that is applied

Check that a limit you declare is actually *read*. The earlier seeker had an attribute for the maximum detection distance, set to a large number, and no line of code anywhere used it. It looked configurable and was decoration. Here, `m_fRangeLimit` is applied in `AcceptTarget` .

Resolve a fixed point once

The earlier seeker called `PointInfo.Init()` in every tick (30 times a second) to find its emitter point. Now it is called once, in `EOnInit` , and only if the prefab set an origin; later only the world matrix is read. For a point that is fixed on the model, once at start is enough. For a point on an *animated bone* it is the other way round: the cached world transform goes stale, and the point has to be resolved again in every frame you want it current. Know which of the two you have.

Night and thermal vision

An infrared laser should be visible to everybody who has night vision, whatever the operator selected. And a sight with night and thermal modes needs to put its filter on the right camera – and take it off the same one.

IN PLAIN WORDS

Each player decides for himself whether he can see infrared. The sight can switch between normal, night and thermal, per camera.

HOW TO HOOK THIS UP IN YOUR OWN MOD

To give your own optic these modes, add `GTG_SOFLAM_VisionModeComponent` to the device and call `SetInputEnabled(true)` when the player starts looking through it and `false` when they stop. If you have your own night-vision optic, call `GTG_SOFLAM_NVState.SetTemporaryNVState(true)` while it is on, and infrared lights become visible for that viewer.

Infrared visibility is a per-viewer fact

Whether *this* client can currently see infrared light is a purely local state. A real IR laser is visible to anyone with their own night vision, independent of what the operator of the device selected. So there is nothing to replicate: each machine keeps one flag, set when the player looks through a night-vision optic. It is a static field of `GTG_SOFLAM_NVState`.

EXAMPLE CODE - FREE TO USE

GTG_SOFLAM_NVState.c (excerpt)

```
class GTG_SOFLAM_NVState
{
    protected static bool s_bTemporaryNV;

    // set by devices: true = this client is looking through a night-vision optic and can see IR
    light
    static void SetTemporaryNVState(bool state)
    {
        s_bTemporaryNV = state;
    }

    static bool IsTemporaryNVState() { return s_bTemporaryNV; }
    static bool CanLocalPlayerSeeIR() { ... } // below
}
```

A STATIC FLAG THAT STAYS ON

The flag is

STATIC

: it holds for the whole client and the rest of the session. In `SetNightVision` it used to be set at the end, behind the check “is there a renderer”. Whenever the method left early, the flag kept its old value, and that player saw *every* IR light without goggles, day and night. The beam of the test tripod showed as a red line to the naked eye. The flag is pure state and has nothing to do with drawing; it is set first, before any early return.

An optional mod without a compile dependency

If the viewer wears the night-vision goggles of another mod, that should count too – but a compile reference to that mod would make yours unbuildable without it. The Core calls it by name at runtime (RHS-StatusQuo's `RHS_IsNV0ff` on the player controller): if the mod is loaded the call returns the real answer, if not it fails and only your own state counts.

EXAMPLE CODE - FREE TO USE

GTG_SOFLAM_NVState.c

```
static bool CanLocalPlayerSeeIR()
{
    if (s_bTemporaryNV)
        return true;

    SCR_PlayerController playerController =
    SCR_PlayerController.Cast(GetGame().GetPlayerController());
    if (!playerController)
        return false;

    ScriptModule scriptModule = GetGame().GetScriptModule();
    if (!scriptModule)
        return false;

    bool nv0ff = true;
    if (scriptModule.Call(playerController, "RHS_IsNV0ff", false, nv0ff)) // false if there is
no such method
        return !nv0ff;

    return false;
}
```

PITFALL

This runs where it is asked – the beam asks every frame while a target exists. The Core does

NOT

remember a failed call; it asks again every time. If you copy the pattern into something hot, remember the first failure and stop asking.

The light itself, `GTG_SOFLAM_LightEntity`, has the gate built in: `SetEnabledWithIRCheck(true)` only switches an IR light on (prefab attribute `m_bIsIR`) if `CanLocalPlayerSeeIR()` says so.

Which modes exist is data

Two attributes of `GTG_SOFLAM_VisionModeComponent`, `m_bAllowNightVision` and `m_bAllowThermal` (both default true), decide which modes a device has at all, and the cycle is built from them: both on gives normal → night → thermal, night only gives normal → night, none switches the whole component off (it hooks up no keys). An older device without thermal needs a tick in the prefab editor, not a class. The mode is the enum `GTG_SOFLAM_EVisionMode` (`NORMAL`, `NIGHT_VISION`, `THERMAL`); `GetOnVisionModeChanged()` returns a `ScriptInvoker` that fires with the component and the new mode, so an optic can update its display without asking every frame. A mode that is switched off on the prefab is quietly taken back to normal when a script asks for it.

The camera-id trap

The thermal filter hangs on a camera *index*. Remember the index in use *when you switch the filter on*, and clear that one when you switch it off. `GetCurrentCameraId()` can have changed in between – the player left the optic quickly, or changed zoom, which in some sights is a different camera. Clearing on the wrong id left the thermal filter on the active camera and the screen stayed white until switching again released it.

EXAMPLE CODE - FREE TO USE

GTG_SOFLAM_VisionModeComponent.c (excerpt)

```
protected void SetThermal(bool state)
{
    if (!GTG_HasRendering())
        return;

    BaseWorld world = GetGame().GetWorld();
    if (!world)
        return;

    if (state)
    {
        m_iThermalCamId = world.GetCurrentCameraId(); // REMEMBER it
        world.SetCameraPostProcessEffect(m_iThermalCamId, m_iThermalSlot,
            PostProcessEffectType.ThermalImaging, GTG_CurrentThermalFilter());
        return;
    }

    int camId = m_iThermalCamId;
    if (camId == -1)
        camId = world.GetCurrentCameraId();

    // clear THE camera we set it on...
```

```

world.SetCameraPostProcessEffect(camId, m_iThermalSlot,
PostProcessEffectType.ThermalImaging, string.Empty);

// ...and, to be safe, the current one too if it differs. Costs nothing where nothing was
set.
int currentCamId = world.GetCurrentCameraId();
if (currentCamId != camId)
    world.SetCameraPostProcessEffect(currentCamId, m_iThermalSlot,
PostProcessEffectType.ThermalImaging, string.Empty);

m_iThermalCamId = -1;
}

```

Night vision uses a vanilla HDR slot. The Core adds one method to the vanilla

`SCR_ScreenEffectsManager` (modded class), `GTG_SOFLAM_SetHDR(ResourceName material, bool enable)`, writing slot/priority 16 (the same as RHS, so that both together do not stack two effects). An **empty** material clears the slot instead of putting the vanilla HDR on it: it used to set the vanilla material without the parameters the game runs it with, the slot stayed occupied, and in daylight the whole picture burned out after leaving night vision.

Smaller measured choices

CHOICE	REASON
Own copies of the thermal filters (<code>GTG_ThermalFilter_WhiteAsHot</code> , <code>_BlackAsHot</code>) with a fixed min/max contrast (<code>AutomaticMinMax</code>)	The stock filter (RHS's <code>AutomaticHistogramPLHE</code>) recomputes its contrast curve periodically, and that shows as short, irregular dropouts of the picture.
Thermal slot <code>m_iThermalSlot</code> = 12	RHS's vehicle thermal uses slot 11. Two effects on the same slot overwrite each other. Pick a number others do not, and note it down.
Night-vision gain levels 0.01 ... 30, start on index 5 (= 1.0)	Taken from the AH-64 mod, where they were flown in.
Mode key J (<code>GTG_SOFLAM_VisionModeAction</code>), not N	In a collection of about 70 mods N did not reach the action at all, not even with shift. Measured in a multiplayer test.

CHOICE	REASON
Other keys: GTG_SOFLAM_ToggleNvThermalAction (night ↔ thermal), _GainUpAction / _GainDownAction (LSHIFT + wheel: night-vision gain or thermal filter), _LaserPowerUpAction / _LaserPowerDownAction (H + wheel)	All six are attributes of the component, so another device can use other action names.
Skip everything visual on the dedicated server (RplSession.Mode() != RplMode.Dedicated)	No render engine, no point doing material work.

The visible beam

Everybody who can see infrared should see the beam. That needed one switch to be turned on and one origin to be decided.

IN PLAIN WORDS

The laser is drawn as a light cone from the device to the target. It is made locally on every machine, and it only appears on other players' screens if you ask for it.

HOW TO HOOK THIS UP IN YOUR OWN MOD

Add `GTG_SOFLAM_BeamComponent` to a device that carries a designator and set `m_sBeamLightPrefab` to `GTG_SOFLAM_IRLaserLight.et`. The component finds the designator on its own entity and draws between the device and the point in the registry.

Components tick only where the entity is owned

By default a component's frame event runs only on the machine that owns the entity. Without asking for more, the beam would be invisible to everybody except the operator – although the target point was replicated correctly. The fix is one override, plus a frame mask that is set permanently rather than tied to the optic: someone who is not aiming has no optic open.

EXAMPLE CODE - FREE TO USE

GTG_SOFLAM_BeamComponent.c (excerpt)

```
class GTG_SOFLAM_BeamComponent : ScriptGameComponent
{
    override event bool OnTicksOnRemoteProxy()
    {
        return true;           // also tick on machines that do not own the entity
    }

    override void OnPostInit(IEntity owner)
    {
        super.OnPostInit(owner);
        m_Owner      = owner;
        m_Designator = GTG_SOFLAM_LaserDesignatorComponent.FindOn(owner);

        SetEventMask(owner, EntityEvent.FRAME);    // permanent: not coupled to who is aiming
    }
}
```

PITFALL

This is the general version of a pattern that costs hours: a component that does its job in `EOnFrame` and works for the owner, but on every other client simply never runs. In Workbench the host is the owner, so nothing shows. The same applies to a vehicle component on a client that does not own the aircraft: its frame events never run there.

The beam knows why it is not drawn

A beam that does not come looks the same at five exit points. The component prints the reason, once per reason, when the same reason has stood for a full second (a short window is normal at start-up, because the entry is created without a position and the first position follows with the next report):

EXAMPLE CODE - FREE TO USE

```
[GTG_SOFLAM] Kein Strahl seit einer Sekunde: kein Designator an dieser Entitaet
[GTG_SOFLAM] Kein Strahl seit einer Sekunde: Ersteller-Id unguelteig -- Traegerentitaet ohne
RplComponent
[GTG_SOFLAM] Kein Strahl seit einer Sekunde: Zielablage nicht gefunden
[GTG_SOFLAM] Kein Strahl seit einer Sekunde: kein Ziel unter dieser Ersteller-Id eingetragen
[GTG_SOFLAM] Kein Strahl seit einer Sekunde: Ziel eingetragen, aber ohne Position
```

In order: no designator on this entity; creator id invalid (carrier without a replication component); registry not found; no target under this creator id; target present but without a position. Only printed while the designator's laser is on.

One origin for everyone

For a while the operator saw the beam from a camera-fixed origin and everybody else from the measured exit of the lens. The camera-fixed one was deliberate – looking along your own beam you would otherwise see just a dot. The price was that the beam came out of the wrong corner: the server log showed offsets of exactly the old constants (`-0.10 / 0.10 / 0.05`), so the code had done what it was told.

The decision was that the beam comes out of the source, for everybody. The special case went away, and there is now one measured aperture – the one that provably leaves from the lowest lens in the outside view. Its offsets are attributes (`m_fEmitterForward` `-0.0736` , `m_fEmitterDown` `0.0253` , `m_fEmitterRight` `-0.0371` , in metres, device coordinates), the start is smoothed with a time constant of `m_fEmitterSmoothTime` = 0.08 s (display only; the reported point stays exact), and for the operator the end of the beam is the freshly computed point (`GetLiveTargetPosition()`) because the crosshair is real time and the reported point is up to one report old.

The cone is trimmed to the target

The light is a spot (`Type Spot` , radius 500 in the prefab). Left alone it ran up to 450 m behind a target at 50 - 250 m and, because the origin sits beside the line of sight, showed up left or right of the target in the depth – it looked like “the laser goes past the target”, while the direction was right all the time (0.04 degrees off, measured). So the radius is clamped to the distance (between 50 and 500 m) every frame, and the near plane scales with it at 1 : 10000.

Local, not replicated

The light entity is created locally on each machine (`SpawnEntityPrefabLocal`), not on a dedicated server. It follows the replicated target point and needs no replication of its own. It is only switched on for those who can see infrared ([the previous page](#)).

A telescopic tripod

Nine parts, three legs that extend by different amounts, a column and a ball head.
The design rule: one calculator, one place where a mistake can show.

IN PLAIN WORDS

A tripod whose legs extend, with a centre column and a ball head. One component computes all the parts, and only a few numbers are sent.

One calculator for all parts

A single component on the root, `GTG_SOFLAM_TripodComponent`, moves all child entities. Nine parts cannot drift apart that way, and a fault has one place to appear. Each child carries a small marker, `GTG_SOFLAM_TripodSegmentComponent`, that says *what* the part is (enum `EGTG_TripodSegment`: `LEG_MID`, `LEG_LOW`, `STEM`, `HEAD`, `FRAME`), along which axis it moves and where it sits when retracted. Three independent quantities, each with its own control: leg extension, centre column, ball head.

The marker uses an enum and not a name search because child entities have no reliable name at runtime. The rest position (`m_vBaseOffset`) is an attribute and is not read from the entity: reading it in the first tick after `OnPostInit` returned garbage on a just-created child, and the legs lay flat at random angles.

Why the structure rises when the legs extend

The feet should stay on the ground while the head goes up. Mathematically those are the same thing seen from two sides, and you have to pick one. Chosen: the legs extend *downwards*, along their own slanted axes, and the structure above rises by exactly the distance the feet move down.

The alternative – hold the feet and push everything above up – fails on geometry. The frame is one rigid piece above all three legs and can only move vertically, but the tubes sit in sleeves inclined about 21°. At 0.5 m of lift every upper tube would run about 20 cm sideways out of its sleeve. It is obvious immediately.

What moves, though, is not the root. The first version lifted the whole entity. The gunner's seat hangs on the root and rose with it, measured:

EXAMPLE CODE - FREE TO USE

```
legs 0.94 | lift 0.5045 | root above ground 0.5045 | gunner 0.5045
```

A compartment slot cannot be moved at runtime. So the root stays on the ground (and has no model any more), the yoke is a part of its own (role `FRAME`), and **every part** gets the lift added to its local offset. Same world positions, different frame of reference:

EXAMPLE CODE - FREE TO USE

GTG_SOFLAM_TripodComponent.c (excerpt)

```
protected float GTG_LegLift()
{
    foreach (GTG_SOFLAM_TripodSegmentComponent segment : m_aSegments)
    {
        if (segment.GetSegment() != EGTG_TripodSegment.LEG_LOW)
            continue;

        vector axis = segment.GetAxis();           // ~21 degrees from vertical: 0.93 m of height
        per metre of extension
        return m_fLegExtension * m_fLegStageTravel * segment.GetStageCount() * axis[1];
    }

    return 0;
}

// in GTG_ApplyNow, for every part: the root stays, the part rises
mat[3] = segment.GetBaseOffset() + offset + hub;
part.SetLocalTransform(mat);
```

Stage travel `m_fLegStageTravel` is 0.29 m in the turret prefab (the script default is 0.28), the column `m_fStemTravel` 0.25 m, the head tilt limit `m_fHeadTiltLimit` 45°. The inner tube (`LEG_LOW`) moves two stages. The limit is the telescope overlap: below 5 cm of overlap a tube holds nothing.

NOTE

That the feet walk outwards while extending is not a bug. A real tripod does the same when extended without re-setting it.

Numbers on the wire

Almost nothing is a transform. Every machine computes the same nine positions from a few numbers. That is less traffic than nine transforms, and it cannot arrive partially: either the state is right or it is wrong everywhere.

- **Leg and column extension** are *signals* on the entity. The user actions `GTG_SOFLAM_TripodHeightAction` (“Set Height Legs” / “Set Height Center”) extend `SCR_AdjustSignalAction`, which already keeps the mouse wheel, the value range, the rounding and the replication to everybody including late joiners. Four hooks pass the value on

(`OnSaveActionData` , `OnLoadActionData` , `OnRplSave` , `OnRplLoad`); miss one and the tripod is wrong for exactly one group of players. The tripod component itself only applies it (`ApplyLegExtension` , `ApplyStemExtension`) and does not send it again.

- **Head angles** (yaw, pitch, roll) go from the server as a broadcast (`GTG_RpcHead`), the base yaw likewise (`GTG_RpcBaseYaw`), and the flag “a device is installed” (`GTG_RpcBestueckt`). When a gunner sits in the turret, the head is not sent at all: `GTG_SOFLAM_TurretHeadComponent` reads the turret's own aim (`TurretComponent.GetAimingRotation()`), or the signals `AimTurnTurret` / `AimElevationTurret` as a fallback) and applies it on every machine. The turret replicates its aim itself; a second transmission would be double traffic and a second truth.
- **Late joiners** fetch head angles, base yaw and the installed flag through `RplSave` / `RplLoad` (extension values come with the actions). Otherwise a fully extended tripod would stand collapsed for everybody who connected later.

There is deliberately **no client-to-server Rpc** in the tripod component. A client does not own the tripod, so a message to the server would be dropped (see [Who may send](#)). Operation goes through user actions that run on the server anyway; the result goes out as a broadcast.

The base always stands level

The tripod keeps its pitch and roll at zero whatever the ground does (`m_fBasePitch = 0` , `m_fBaseRoll = 0`); only the yaw comes from the placement. Taking over the slope of the ground meant the head tilt needed for a target changed just by panning, and the 45° clamp was reached much earlier in some directions. Like a real tripod with a spirit level, the legs make up the difference.

Every leg follows its own ground

On sloped ground one common extension makes at least one leg hang in the air or stand in the ground. The fix keeps the frame rigid and level, but gives each leg its *own* extension along its *own* unchanged axis – exactly what a telescopic leg with a fixed hinge angle does.

The first version asked for the ground height once, at the leg's mount point. But the leg extends along an axis inclined about 21°, so its foot lands **20–35 cm to the side** of that point, and on a slope the ground is a different height there. The answer is a short fixed-point iteration: foot position from the current extension, ground at that foot, new extension, repeat. Four passes leave about 2 % error, and it converges as long as the slope times the tangent of the leg tilt (0.38) is below 1, which holds well past a 45° slope.

EXAMPLE CODE - FREE TO USE

GTG_SOFLAM_TripodComponent.c (excerpt)

```
// per leg, from GTG_BeinFraktionenBeiHub (shortened)
float fraktion = 0;
```

```

for (int durchlauf = 0; durchlauf < 4; durchlauf++)
{
    // where does the foot land with this extension?
    vector fussWelt = montageWelt - achseWelt * (fraktion * maxReichweite);

    // what is the ground there? (a trace, not the height map - see below)
    float bodenY = GTG_BodenY(welt, Vector(fussWelt[0], montageWelt[1] + 0.3, fussWelt[2]));

    // the new extension that reaches it
    fraktion = (montageWelt[1] - bodenY) / (achseLokal[1] * maxReichweite);
    if (fraktion < 0)
        fraktion = 0;
}

```

If a leg cannot reach the ground at full length, the *common* height is lowered in steps of 5 % of the wanted lift (at most 40 steps) until all three legs are within range. The lowest leg decides the maximum height, as on a real tripod.

EXAMPLE CODE - FREE TO USE

GTG_SOFLAM_TripodComponent.c (excerpt)

```

while (schlechtesteFraktion > 1.0 && effektiverHub > 0.0001 && versuch < 40)
{
    effektiverHub -= wunschHub * 0.05;
    if (effektiverHub < 0)
        effektiverHub = 0;

    schlechtesteFraktion = GTG_BeinFraktionenBeiHub(baseMat, effektiverHub, achsen,
montagepunkte,
        maxReichweite, welt, fraktionen);
    versuch++;
}

```

This geometry is not recomputed for every aim update. It is only measured again when the wanted lift, the placement or the yaw changes; before that, aiming cost frame rate, because the tripod re-applied itself every frame and each pass fired a dozen traces.

HEIGHT MAP VERSUS THE REAL SURFACE

`GetSurfaceY()` is the height map. The surface a player actually stands on can be higher. In one measurement the player stood 0.55 m and the tripod root 0.63 m *above* the height-map value. If the feet or a door point follow the height map, they end up inside the surface. The Core measures the ground with a downward trace (`SCR_TerrainHelper.GetTerrainY` with a trace on the layers `Terrain | Static`, the tripod's own parts excluded; where the trace hits nothing it falls back to the height map).

A watchdog, honestly named

The device sitting on the ball head moves only on events: mounting, height change, aiming. If something else moves it (a weapon slot, the inventory, physics) nothing corrects it, and the log looks healthy while the device lies at the player's feet. A watchdog compares actual against expected position every 250 ms (tolerance 5 cm), *reports* it once per incident with both positions, and resets it. A summary follows every five seconds while corrections continue.

EXAMPLE CODE - FREE TO USE

format of the report (values are examples)

```
[GTG_SOFLAM][WACHHUND] Aufgesetztes Geraet 0.8 m von seinem Platz -- ist <...> soll <...>
(Kugelkopf <...>). Vershoben hat es NICHT diese Komponente; wird zurueckgesetzt.
```

NOTE

The report stays on even when the watchdog corrected the position. As long as it appears, there is a cause that has not been found. It is a bandage, not a cure.

Assembling and the gunner seat

Build the turret from two parts and let the engine do the spawning. And a fake door that has to be placed on real ground.

IN PLAIN WORDS

Two items become one turret in a single engine action; the player sits in a seat; the seat needs a door, and the door must stand on real ground.

Two parts, one engine action

The folded tripod (`GTG_SOFLAM_Tripod_Carried.et`) is a vanilla multi-part deployable. You drop it, look at it, and “Assemble” builds it together with a SOFLAM from your inventory (`PEQ1_SOFLAM.et` , part name “SOFLAM Designator”). A ghost shows the result first. Both parts are consumed and the engine spawns `GTG_SOFLAM_Tripod_Turret.et` . The component on the carried tripod:

EXAMPLE CODE - FREE TO USE

`GTG_SOFLAM_Tripod_Carried.et` (excerpt)

```
SCR_MultiPartDeployableItemComponent "{61B46B9C8F38F759}" {
  m_aVariants {
    SCR_DeployableVariantContainer "{6A48B5C100000310}" {
      m_sReplacementPrefab "{6A48B5E000005010}Prefabs/Items/Tripod/GTG_SOFLAM_Tripod_Turret.et"
      m_sPreviewObject "{6A48B5E000005020}Prefabs/Items/Tripod/GTG_SOFLAM_TripodDeployPreview.et"
      m_aRequiredElements {
        SCR_RequiredDeployablePart "{6A48B5C100000311}" {
          m_sPrefab "{F2A55A212EBAF03F}Prefabs/Items/SOFLAM/PEQ1_SOFLAM.et"
          m_sPartName "SOFLAM Designator"
        }
      }
    }
  }
  m_fMaxAllowedTilt 35
}
```

“Pack Up Tripod” gives both back, separately. Vanilla solves that with a *second* multi-part component on the assembled object: the replacement prefab is the main part, `m_aAdditionalPrefabs` are the others (copied from the vanilla M3 tripod with the M2HB):

EXAMPLE CODE - FREE TO USE

`GTG_SOFLAM_Tripod_Turret.et` (excerpt)

```
SCR_MultiPartDeployableItemComponent "{6A48B5C000006210}" {
  m_bRemoveAllItemsWhenDeleted 1
  m_aVariants {
    SCR_DeployableVariantContainer "{6A48B5C000006211}" {
      m_sReplacementPrefab "{6A48B5E000005009}Prefabs/Items/Tripod/GTG_SOFLAM_Tripod_Carried.et"
      m_aAdditionalPrefabs {
```

```

SCR_AdditionalDeployablePart "{6A48B5C000006062}" {
  m_sPrefab "{F2A55A212EBAF03F}Prefabs/Items/SOFLAM/PEQ1_SOFLAM.et"
}
}
m_bUsePartRotationAndPosition 1
}
}
m_eSurfaceObservationBehaviour DISMANTLE
}

```

WHAT CHANGED

The earlier route had two steps: deploy the tripod, then “Mount Designator”, and “Collect Designator” to take it off again. Both actions are gone from the shipped prefabs. Collecting left a tripod with no device on it, and in that state “Use SOFLAM” disappears. The classes (`GTG_SOFLAM_MountAction` , `GTG_SOFLAM_PickUpAction` , `GTG_SOFLAM_SwapPrefabAction` , `GTG_SOFLAM_UseAction`) are still in the script folder; no shipped prefab uses them.

Why a weapon spawned by script arrives empty

The turret holds the device in its weapon slot: `WeaponTemplate` points to `PEQ1_SOFLAM_Mounted.et` , and the engine creates it on every machine. The route through a script first (spawn the weapon prefab, put it in the slot) produced a device that existed for the server and for nobody else. The cause was not the spawn but the prefab: the mounted device inherited `Weapon_Base` , an **abstract** base with a weapon animation component but no animation graph. Without a graph the entity initialisation aborts and the engine returns an empty shell: no model, no components, and no replication component. Nothing to replicate, so no client ever saw it.

EXAMPLE CODE - FREE TO USE

from the server log of 2026-09-06 (shortened)

```

SpawnEntityPrefab {6A48B5E000005012}PEQ1_SOFLAM_Mounted.et
Create entity ('GenericEntity')          <- without a model
ANIMATION (E): anim controller initialization data is invalid (graph , node MasterControl,
instance )

Server  slot 1 | weapon 1 | designator false | Rpl false | children 9
Client  slot 1 | weapon 0 | prefab -          |          | children 8

```

The fix is to give the device the animation set the turret already runs. Weapon and turret must use the same set, or grips and seat do not match:

EXAMPLE CODE - FREE TO USE

PEQ1_SOFLAM_Mounted.et (excerpt)

```

WeaponAnimationComponent "{60B4EA76EB15F6E0}" {
  AnimGraph "{C727601154428730}Assets/Weapons/HeavyWeapons/workspaces/M2HB_tripod.agr"
  AnimInstance "{26749793D7EDEB6D}Assets/Weapons/HeavyWeapons/workspaces/M2HB_Weapon_tripod.asi"
}

```

```
StartNode "WeaponMasterControl"  
AutoVariablesBind 1  
}
```

NOTE

Rule that came out of it: if a freshly created weapon has no components on a dedicated server, look at its inheritance chain and its animation graph before you blame the server or the spawn path.

The legacy route: re-parent the same entity

The code that mounts a device by hand is still in `GTG_SOFLAM_MountAction`, and the technique is worth knowing. The carried device leaves the inventory and becomes a child of the ball head. It keeps *everything*: laser code, vision mode, a running target report, its replication id. The obvious alternative – delete the carried device and spawn a “mounted” prefab – is simpler, but throws exactly that state away and needs a second prefab.

EXAMPLE CODE - FREE TO USE

GTG_SOFLAM_MountAction.c (excerpt)

```
item.SetWorldTransform(worldMat);           // 1. world position of the seat  
first  
  
// 2. then attach, and let the engine convert that world transform to a local one  
support.AddChild(item, -1, EAddChildFlags.RECALC_LOCAL_TRANSFORM);  
  
SetItemPhysicsActive(item, false);          // SimulationState.NONE: otherwise  
it falls off the head
```

PITFALL

The order matters. With `AUTO_TRANSFORM` first the transform you just set is immediately overwritten; with `NONE` the device never follows the tripod. `RECALC_LOCAL_TRANSFORM` after setting the world position is the combination that does both.

Do it on the authority only (`if (m_RplComponent && m_RplComponent.IsProxy()) return;`). If a proxy also did it, the device would hang on the tripod on one machine and not on the others. And switch the physics off with `SCR_PhysicsHelper.ChangeSimulationState(item, SimulationState.NONE)`, not just to inactive: an inactive body stays in the collision world and its position still belongs to the physics, not to the hierarchy.

Aiming from the gunner seat

“Use SOFLAM” puts the player into the turret seat. `GTG_SOFLAM_TurretGetInAction` derives from the vanilla `SCR_GetInUserAction` and not from a modded class, because the vanilla action sits on every vehicle and turret in the game. It hides the action when no device is installed (an unavailable entry would look like a missing permission; the reason text is `#AR-UserAction_SeatObstructed`), and selects the turret weapon a moment after getting in if the engine did not.

The turret is set up so that you look around freely and the aiming axis follows the mouse only while looking through the optic: `CanAimOnlyInADS 1`, limits `LimitsHoriz -180 180` and `LimitsVert -45 45` (also for prone), and `UseWeaponSights 1` so that the optic of the device counts. The ball head follows the turret locally ([the tripod page](#)).

RADIANS, NOT DEGREES

The engine call `SetAimingAngles` takes

RADIANS

. It was given degrees from the tracking code. Both axes ran into their stops: elevation read exactly 45.000, the stop value, and the mount stood 180° off. From outside it looked right, because the ball head followed through a second call that already used radians. Measured: sight axis off by 133.3° before, 0.13° after.

The seat needs a door

A vehicle-style seat needs a door the engine can resolve. Without one the get-in check finds no valid door index and reports the seat as obstructed – “Obstructed” on completely free ground. A *fake door* with an entry, an exit and an exit-teleport point fixes that. Do not override the door index lookup to a constant: the stock method does a real query for exactly the case you care about (the local player getting in). An earlier override that returned `-1` replaced that query for every caller, the fake door was never looked at, and it was the real cause of “obstructed” on flat ground.

EXAMPLE CODE - FREE TO USE

`GTG_SOFLAM_Tripod_Turret.et` (excerpt)

```
CompartmentDoorInfo "{6A48B5C000006050}" {
    ContextName "tripod_head"
    EntryPositionInfo PointInfo "{6A48B5C000006051}" {
        Offset 0 0 -1.5
    }
    ExitPositionInfo PointInfo "{6A48B5C000006052}" {
        Offset 0 0 -1.5
    }
    ExitTeleportPositionInfo PointInfo "{6A48B5C000006053}" {
        Offset 0.5 0 -1.5
        Angles 0 -30 0
    }
}
```

```

GetInTeleport 1
GetOutTeleport 1
FakeDoor 1
StanceChangeOnExit "Change from turret height"
}

```

A door point on a hillside

The fake door's points are fixed offsets from the root, 1.5 m behind it. That silently assumes the ground there is as high as at the root. On a slope it is not, and the engine's native check at that point can answer “obstructed” although the player stands right in front of it.

So place the points at runtime on the ground, keeping direction and angle from the prefab and adjusting only the height. And use the *real* surface for that height, with a little air (5 cm), as the previous page warns: a point exactly on the surface or inside it counts as blocked. The point is changed with `PointInfo.Set(owner, boneName, matrix)`, which the engine header allows at runtime.

EXAMPLE CODE - FREE TO USE

GTG_SOFLAM_TripodComponent.c (excerpt, log lines left out)

```

protected void GTG_ZiehePunktAufBoden(PointInfo punkt, IEntity owner, vector ownerMat[4],
BaseWorld welt)
{
    vector lokal[4];
    punkt.GetLocalTransform(lokal);

    vector weltMat[4];
    Math3D.MatrixMultiply4(ownerMat, lokal, weltMat);

    // the real surface by trace, plus 5 cm of air; the trace starts 1 m above the point
    float boden = GTG_BodenY(welt, Vector(weltMat[3][0], weltMat[3][1] + 1.0, weltMat[3][2])) +
0.05;
    float differenz = boden - weltMat[3][1];

    if (Math.AbsFloat(differenz) < 0.02)
        return; // no write without a reason

    lokal[3][1] = lokal[3][1] + differenz;
    punkt.Set(owner, punkt.GetBoneName(), lokal);
}

```

It is applied to the entry, exit and exit-teleport point of door 0, on every machine, from the same code that places the legs, and only when the geometry key changed. No RPC is needed because the terrain read is the same everywhere.

NOTE

The diagnostic that made this findable hung on the moment of refusal: the log (`GTG_LOG_SPERRE` , off by default) writes the reason text, the player and root positions with the ground under each, and the entry point's height above the ground. Without it the question “why obstructed?” has no answer, because the verdict comes from a native function.

“Too steep” without guessing

When assembly refuses with “too steep” on a meadow that looks flat, do not replace the check – measure it. `GTG_SOFLAM_TripodAssembleAction` derives from the vanilla `SCR_DeployMultiPartInventoryItemAction` , overrides `SetCannotPerformReason` , and repeats the base class's trace (throttled to every 2 s) *only* when exactly the too-steep text was set. Anything else that makes the action fail would be noise in the log. The limit is `m_fMaxAllowedTilt 35` degrees on the carried prefab.

EXAMPLE CODE - FREE TO USE

the measurement line

```
[GTG_SOFLAM][ZUSAMMENBAU] Spurpunkt <position> | Getroffen <entity> | Normale <vector> | Neigung  
<angle> Grad | Grenze 35
```

Dedicated-server pitfalls

Each of these cost real time. They have in common that the single-player run looked fine.

IN PLAIN WORDS

The short list of things that worked in Workbench and failed on a server.

SYMPTOM	CAUSE	FIX
Nothing happens on the server, no error	The client does not own the entity, so its server-bound Rpc is dropped silently	Send through a relay on the player's character (<code>GTG_SOFLAM_Relay</code>)
The device is invisible to everybody but the host	The mounted prefab inherited an abstract weapon base with no animation graph; initialisation aborted and the engine returned an empty shell with no replication component	Give it the animation graph the turret uses (<code>M2HB_tripod.agr</code>), and log slot, weapon, children and role on both machines to see the difference
A beam or effect appears only for the operator	The component ticks only on the owning machine	Return true from <code>OnTicksOnRemoteProxy()</code>
Aiming flies 180° off and freezes at a limit	An engine call that takes radians (<code>SetAimingAngles</code>) was given degrees; elevation read exactly the stop value	Convert once, at the call
The mount moves in steps while the picture runs smoothly	Aiming hung on a throttle that was meant for network updates	Throttle the <i>sending</i> , not the local update
The registry system is never found (<code>Zielablage nicht gefunden</code>)	No update point, or not listed in the systems config	Add a point and the config entry

SYMPTOM	CAUSE	FIX
Stale targets steer missiles	Entries nobody removed	Expiry on last report (5 s); publishers re-report every 2 s
A missile turns to the world origin for a moment	A target created but not yet reported sits at the zero vector	Skip the zero vector when reading
Everybody sees IR lights without goggles	A static flag that was not reset on an early return	Set the flag first, before any check
The whole script module disappears	A compile error in <i>any</i> loaded mod, or a name clash	Prefix every class, and keep foreign mods out of the dependency chain
Log fills with thousands of lines	A diagnostic switch left on after a test run	Default every switch off (the Core's are <code>GTG_DIAG</code> / <code>GTG_LOG_*</code> constants), and say in the comment what turns it on

How to diagnose

Put one searchable tag on every log line (`[GTG_SOFLAM]` in the Core, `[GTG_AH64]` in the AH-64 mod) and write the *role* of the machine into every network line: server, owner, proxy without owner. Then “SENDE without WIRKT” (send without effect) is visible at a glance, and so is the missing direction.

NOTE

Measure before changing. In most of these cases the first explanation was wrong, and the second one came from the log.

Integrating a helicopter or aircraft

Adding laser guidance to an aircraft is mostly not about the laser. It is about racks that do not launch, seats that are called something else, keys that were renamed, and state that only one machine knows. This is the list of things that went wrong in the AH-64 and AH-1Z mods, in the order you will meet them.

IN PLAIN WORDS

What went wrong when laser-guided missiles were put on helicopters: weapon racks, mode keys, phantom explosions, the missile camera, and how to test it.

WHERE THIS FITS

The how-to for the Core part (code holder, dials, injector, seeker) is on [Aircraft, vehicles and turrets](#). This page is what went wrong around it. The classes named here are from the AH-64 mod (`GTG_AH64_*`); the base classes `WCS_Armament_*` belong to WCS Armaments, not to the Core.

What an aircraft needs

Five small pieces, each with one job. None of them knows the designator: they only know the registry and a code. **All of them are built by the aircraft mod; the Core provides only the registry.**

PIECE	SITS ON	JOB	CLASS IN THE AH-64 MOD
Code holder	The turret	Holds the aircraft's four-digit code, replicates it on every change and hands it to late joiners	<code>GTG_AH64_LaserCodeComponent</code>
Cockpit code dials	Four user actions, positions on the vehicle	Let the crew change the code; the action and the holder must be findable from each other	<code>GTG_AH64_CockpitLaserTuneAction</code> (<code>m_iLaserCodeDigit</code> 0..3)
Seeker	The missile	Asks the registry for the nearest target with the aircraft's code	<code>GTG_AH64_LaserSeekerComponent</code> , a subclass of <code>WCS_Armament_LaserSeekerComponent</code>

PIECE	SITS ON	JOB	CLASS IN THE AH-64 MOD
Flight computer	The missile	Owens the attack modes and flies them (four: DIR, LOBL, LO, HI)	GTG_AH64_LaserFlightComputerComponent
Code injector	The turret	Copies the code onto every missile as it leaves the rack	GTG_AH64_LaserCodeInjectorComponent



NOTE

Give every class an aircraft prefix. Two aircraft mods then run side by side and keep separate codes, and a name clash cannot take the script module down. The AH-64 and AH-1Z mods carry their own copies of the same scripts under `GTG_AH64_` and `GTG_AH1Z_`.

Weapon racks

The upper row that never leaves the rail

A missile pod can carry its rounds in two rows. If the upper row sits so close to the pod body that the engine refuses to apply the launch impulse, those missiles leave the rail carrying only the aircraft's own speed and fall. The signature is unmistakable: of sixteen missiles exactly eight drop, and it is always the same eight, because the launcher serves the upper row of every pod first.

The fix is one attribute on the pod's weapon component: push the missile clear *before* the engine launches it. Measure it. The AH-64 flight computer writes the start state of each missile once (including the velocity *vector*, which tells “never launched” from “launched and stopped” – its length alone cannot).

EXAMPLE CODE - FREE TO USE Pod_M299_AH64D.et (excerpt)

```
WCS_Armament_WeaponComponent "{629D11C992611C79}" {
  m_fLaunchOffset 2      // the base value in Pod_M299.et is 0 and leaves the upper row without
  impulse
}
```

	UPPER ROW	LOWER ROW
Launch speed before	1.6 - 2.1 m/s, then free fall	77 - 93 m/s

	UPPER ROW	LOWER ROW
Launch speed after	78 - 94 m/s	78 - 94 m/s

The AH-1Z mod's own pod prefab (`Pod_M299_AH1Z.et`) sets the same value, with a comment that exactly half of the 16 missiles fall off without it. If your pod is derived from the stock M299, check it.

How high the tube sits

On a helicopter standing on the ground the tubes are under a metre above the terrain (0.9 m on the AH-64). A flat attack mode then grazes the ground in front of the aircraft. That is geometry, not a fault in the rack, and it does not occur in flight. Test from the air before you chase it.

Override or new variant

The first AH-64 mod built a complete new vehicle variant. Its code dials and the third munition choice then did *not* appear on the Apache that was already in the test world. The fix was to make the turret and pod prefabs claim the same resource identity (GUID, through the `.et.meta` file) as the WCS originals: the unchanged vehicle prefab already refers to them by GUID, and that reference now points at the mod's version. Read the cost, too: such an override has to be compared with the original at every update of the base mod; the AH-64 mod keeps a copy of the original turret base under its own GUID for that reason.

Seats and turret controls

The seat has a name

A component that registers a key only while the local player sits in a named seat has to be told the right name. It differs between aircraft: in the AH-64 the weapon seat is `gunner` , in the AH-1Z `copilot` . Read it from the vehicle prefab, do not assume. And do not confuse the compartment name with the interaction context a user action hangs on ([step 5](#)).

Do not keep a second counter

The first version counted every press of the mode key to know which attack mode was selected. It drifted: WCS only switches the mode while the missile weapon is selected, so every press with the rocket pod or the cannon in hand moved the private count one place ahead of the screen. The report was “HI flies like DIR, LO flies highest” – everything shifted by one. Read the mode WCS itself keeps; it is public and the server's copy is kept up to date.

```
int GetCurrentIndex()
{
    IEntity owner = GetOwner();
    if (owner)
    {
        WCS_Armament_LoadoutComponent loadout =
        WCS_Armament_LoadoutComponent.Cast(owner.FindComponent(WCS_Armament_LoadoutComponent));
        if (loadout)
            return loadout.GetDeliveryModeIndex();    // the mode WCS shows on the screen =
the mode that has to fly
    }

    return m_iCurrentIndex;    // own counter only as a fallback
}
```

Input actions get renamed

WCS_Armaments 8.3.0 routes its own action name for cycling the attack mode to the vanilla turret action `TurretWeaponNextFireMode` and registers only that. A listener on the old name stayed silent: the screen switched modes, the server kept flying the first one, and nothing was logged. Resolve the name through the helper WCS uses itself, so the next change does not break you again.

```
m_InputManager.AddActionListener(WCS_Armament_InputBindings.Action("WCS_Armament_CycleWeaponFire
Mode"), EActionTrigger.DOWN, OnCycle);
```

PITFALL

Re-check every hook after a WCS update. An `override` of a method that no longer exists does not just fail your mod: it takes the whole script module down for everybody.

Where the code runs

The key is handled on the client, the missile flies on the server. Anything the flight logic needs – the code, the mode – has to be replicated or sent to the server (the AH-64 mod's mode sync sends from client to server). A purely local variable is invisible to it, and in Workbench you will not notice, because client and server are the same process there.

The same split hides a second trap: on a client that does *not* own the aircraft, the frame events of a vehicle component never run. Anything that has to move every frame on every machine needs its own entity, or a call queue (the AH-64 mod's missile camera is such an entity; its seeker-camera component keeps a call-queue clock as a replacement).

The laser code

- **Replicate it on every change** and hand it to late joiners. A code that is only sent when something is toggled leaves a pilot reading a stale number on his screen.
- **The seeker takes the nearest target with its code.** Two lasers with the same code, one of them next to the aircraft, send every missile to the nearer one. Give a test emitter its own code (the AH-64 and AH-1Z mods ship a test emitter that publishes a fake SOFLAM into the registry), and remove always-on test devices from the test world.
- **Show the code where the crew looks.** A text overlay follows the head; the AH-64 mod moved the code onto a cockpit screen instead.

The missile in flight

A phantom explosion

WCS remembers the last terrain contact of a missile and, when the fuse fires, uses it as the explosion position without checking how old it is. A lofted missile that brushes a hillside on the way down then shows its explosion where it touched, up to 215 m before the warhead really detonates (measured: contact at 2315/98/2353, detonation 0.45 s later at 2507/150/2265). The fix is a guard on the method that publishes the impact: if the remembered contact is far from the missile, use the missile.

EXAMPLE CODE - FREE TO USE GTG_AH64_MissileManeuverability.c (excerpt; the log lines are left out)

```
modded class WCS_Armament_MissileMovementComponent
{
    override protected void PublishImpact IEntity hitEntity, vector position, vector normal,
    ResourceName effectOverride, bool useRecordedContact = false)
    {
        // At about 500 m/s a missile covers up to 8 m per frame; a real contact is far closer
        than 25 m.
        if (useRecordedContact && m_Projectile && GTG_IsOurHellfire() &&
        vector.Distance(position, m_Projectile.GetOrigin()) > 25)
        {
            super.PublishImpact(null, m_Projectile.GetOrigin(), vector.Up, effectOverride,
            false);
            return;
        }

        super.PublishImpact(hitEntity, position, normal, effectOverride, useRecordedContact);
    }
}
```

NOTE

Scope every change to the shared missile class with a test that only your own missile passes (here `GTG_IsOurHellfire()`). The class is the base of every missile in WCS.

A camera that follows the missile

A missile camera has to be its own entity, because of the ownership rule above. Update it *right after the missile has moved in the same frame*. Updated before, it trails: in the dedicated-server tests the camera hung 10 to 28 m behind the missile (one run measured 10 - 19 m, another 17 - 28 m) instead of one metre in front of it. The AH-64 camera is a `ScriptCamera` subclass that sets its world matrix in its **own** post-frame event, after the movement, and then lets the base class pass exactly that fresh matrix to the engine; there is no order race with the vehicle component.

EXAMPLE CODE - FREE TO USE `GTG_AH64_SeekerCameraDisplayComponent.c` (excerpt; the body is shortened)

```
class GTG_AH64_MissileCamera : ScriptCamera
{
    protected IEntity m_Ziel;           // the missile
    protected vector m_vLokalPosition;  // where on it the camera sits

    override protected void EOnPostFrame(IEntity owner, float timeSlice)
    {
        if (m_Ziel)
        {
            // ... set this camera's world matrix from the missile's CURRENT world matrix ...
        }

        super.EOnPostFrame(owner, timeSlice); // the base hands this matrix to the engine
    }
}
```

Testing it

CHECK	WHY
Run a real dedicated server and a client	Workbench hides ownership, replication and the key-versus-flight split
Fire from the air, at 30 m or more	A parked helicopter adds a ground-contact effect that has nothing to do with your work
Fire every mode from every rack row, left and right	Rack faults show as a pattern: half the rounds, or one side

CHECK	WHY
Compare the peak height of each mode	The modes must differ in the order the screen shows them
Press the mode key with another weapon in hand, then select the missile again	Catches any private counter that drifts from the screen
Use a unique code, and no other laser nearby	The nearest matching target wins
Tag every log line, write the role of the machine into it, then switch the logging off	A switch left on fills the log with thousands of lines (one HUD component once wrote about 500 lines a second)

NOTE

Measure before changing. Most of the faults on this page had a plausible first explanation that was wrong. The right one came from the server log.

Working examples

Two aircraft that use these pieces on the Workshop:

- [SagesAH1Z – SOFLAM Integration](#) – the AH-1Z by Sage, with laser guidance on its copilot seat.
- [AH-64 SOFLAME Integration](#) – the AH-64D with missile camera, TADS screen and attack modes.

Both list the SOFLAM Integration Core as a dependency.