# A Robust, Local-First Architecture for Integrating Seneye USB Sensors (PAR, PUR, LUX) with Home Assistant

## Introduction

This report presents a definitive, production-grade strategy for integrating the full suite of Seneye USB Device (SUD) sensors—including the critical Photosynthetically Active Radiation (PAR), Photosynthetically Usable Radiation (PUR), and Lux metrics—directly into Home Assistant. The monitoring of complex aquatic or horticultural ecosystems demands the highest degree of data fidelity and system reliability. Common integration approaches that rely on network protocols such as MQTT, while flexible, introduce multiple potential points of failure, including network latency, broker instability, and cloud service dependencies. Recognizing the inherent fragility of these network-dependent protocols for critical infrastructure monitoring, this document will architect a superior solution founded on direct USB communication. This local-first approach guarantees maximum reliability and data integrity by systematically eliminating network and cloud dependencies. By establishing a direct, physical connection between the monitoring device and the automation host, this architecture creates a resilient and powerful monitoring system suitable for the most demanding advanced aquarium and horticultural applications.

## Section 1: Foundational Concepts: Understanding the Data and the Device

A robust technical solution is built upon a solid theoretical foundation. Before architecting the software integration, it is imperative to establish a comprehensive understanding of both the scientific principles behind the data being measured and the technical landscape of the hardware providing that data. This section provides a detailed examination of the advanced lighting metrics essential for photosynthetic life and analyzes the various data access methodologies available for the Seneye USB Device (SUD), culminating in the justification for a direct, local-first communication strategy.

### 1.1 Demystifying Advanced Aquarium & Horticultural Lighting Metrics

The quality and quantity of light are among the most critical environmental parameters for photosynthetic organisms, such as corals and aquatic plants. While basic metrics can provide a general sense of brightness, a sophisticated understanding requires differentiating between several distinct measurements. The Seneye device, particularly the Reef model, is capable of capturing these nuanced metrics, which are essential for optimizing growth, health, and coloration.1

* **Photosynthetically Active Radiation (PAR):** PAR designates the spectral range of solar radiation from 400 to 700 nanometers (nm) that photosynthetic organisms are able to use in the process of photosynthesis.2 It is a measure of the  
  *quantity* of light, specifically the total number of photons within this range that strike a given surface area over a specific time period. This is often expressed as Photosynthetic Photon Flux Density (PPFD), with units of micromoles per square meter per second (μmol/m2/s). However, PAR is agnostic to the wavelength of the photons it counts; a photon at 550 nm (green) is counted the same as a photon at 450 nm (blue) or 660 nm (red).3 This is a significant limitation, as not all photons within the PAR range are equally effective at driving photosynthesis. A light source could produce a very high PAR value consisting primarily of green light, which is largely reflected by many plants (hence their green appearance), making it of little use for growth.4 Therefore, relying on PAR alone can be a misleading indicator of a light's effectiveness.
* **Photosynthetically Usable Radiation (PUR):** PUR represents a more refined and biologically relevant metric. It is the *subset* of PAR that is actually absorbed and utilized by an organism's specific photosynthetic pigments.5 The primary pigment, chlorophyll, has absorption peaks in the blue (approximately 400-500 nm) and red (approximately 620-700 nm) parts of the spectrum.2 Accessory pigments, such as peridinin found in the zooxanthellae symbiotic with corals, can extend this absorption range into the green spectrum (up to about 550 nm).5 PUR, therefore, represents the true quality and effectiveness of a light source for a specific organism. The Seneye device is uniquely capable of providing an estimated PUR value because it measures not just the total PAR but also the spectral distribution of the light, allowing it to calculate the percentage of PAR that falls within the usable ranges.4 For advanced ecosystem management, optimizing for PUR, not just PAR, is the primary goal.
* **Lux:** Lux is a unit of illuminance, which measures the intensity of light as perceived by the human eye.9 Human vision is most sensitive to light in the green-yellow part of the spectrum (around 555 nm) and is less sensitive to blue and red light. Consequently, a light source rich in green wavelengths will register a very high Lux value, even if it has a low PUR value and is ineffective for photosynthesis.9 Lux is an aesthetic metric, useful for determining how bright a tank appears to a human observer, but it is not a reliable or scientifically valid metric for assessing the photosynthetic potential of a lighting system.
* **Kelvin (Correlated Color Temperature - CCT):** The Kelvin rating describes the color appearance of a light source, measured in degrees Kelvin (K). It compares the color of the light to that of a theoretical black-body radiator at a specific temperature. Lower Kelvin values (e.g., 2700K) appear "warm" (yellowish/reddish), while higher values (e.g., 10000K or 20000K) appear "cool" (bluish). In the context of aquarium lighting, the Kelvin rating is an aesthetic indicator and provides a rough proxy for the spectral balance of the light.10 For example, a high-Kelvin light is expected to have a higher proportion of its energy in the blue part of the spectrum, which is beneficial for coral fluorescence and photosynthesis.

To clarify the distinct roles of these metrics, the following table provides a comparative summary.

| Metric | Definition | Spectral Range | Primary Use Case | Key Insight |
| --- | --- | --- | --- | --- |
| **PAR** | A quantitative measure of all photons within the photosynthetic range. | 400–700 nm | General measure of light intensity for plant/coral growth. | High PAR is not always better; spectral quality is critical. |
| **PUR** | The subset of PAR photons that are actually absorbed by photosynthetic pigments. | Primarily blue (400-500 nm) and red (620-700 nm) | True measure of a light's effectiveness for a specific organism. | The most important metric for optimizing growth and coloration. |
| **Lux** | A measure of light intensity as perceived by the human eye. | Peaks at ~555 nm (Green) | Measuring brightness for human vision and aesthetics. | Poor indicator of photosynthetic potential; high Lux can have low PUR. |

### 1.2 The Seneye USB Device (SUD) and the Local Control Landscape

The Seneye platform offers several pathways for data access, reflecting an evolution from a cloud-centric model to one where a technically proficient community has pioneered more robust, local-first solutions. An analysis of these pathways reveals a clear architectural choice for achieving the reliability demanded by the user.

* **Official Access Methods:** Seneye provides two primary official methods for device interaction: the Seneye Connect Application (SCA), a Windows-based program, and the Seneye Web Server (SWS), a dedicated hardware device.11 Both are designed to read data from the USB device and upload it to Seneye's cloud service. While necessary for initial device registration and slide activation, relying on these components for a permanent automation solution is undesirable. It introduces dependencies on either a dedicated Windows machine or proprietary hardware, moving away from a streamlined, software-defined system running on a platform like a Raspberry Pi.
* **Alternative Data Pathways:** The user community has documented and developed tools for two main alternative access methods:
  1. **Cloud API (api.seneye.com):** This RESTful API allows users to retrieve the last reading that was successfully uploaded to Seneye's servers.13 While this decouples the data retrieval from the SWS/SCA, it fundamentally relies on a stable internet connection and the availability of Seneye's cloud infrastructure. For a system where reliability is paramount, introducing an external cloud dependency is a significant architectural flaw and is unsuitable for our goals.
  2. **Local Data Exchange (LDE):** This feature allows a configured SWS or SCA to issue an HTTP POST request containing the latest sensor readings to a user-specified URL on the local network.10 The data is securely transmitted using JSON Web Tokens (JWT).10 While this keeps the data within the local network, it still necessitates the continuous operation of the official SWS hardware or SCA software, acting merely as a local data forwarder. This adds a layer of network communication and an extra piece of required hardware/software, both of which are potential points of failure.
* **The Definitive Solution: Direct USB Communication:** The most robust and direct method for data acquisition is to communicate with the Seneye USB Device itself. Seneye officially supports this approach for developers by providing the seneye/SUDDriver repository.1 This repository contains documentation and source code examples in C and C# that detail the low-level protocol for interacting with the SUD over USB. This establishes a clear precedent for direct, local control, completely bypassing the cloud, the SWS, and the SCA.
* **The Pythonic Bridge: pyseneye:** The existence of the SUDDriver paved the way for community developers to create a Python-native implementation of this direct communication protocol. The mcclown/pyseneye library is the cornerstone of the proposed solution.11 It is a pure Python library that leverages the  
  PyUSB package to implement the commands and data parsing logic documented in the SUDDriver. This library is the critical component that unlocks a truly local, reliable, and software-only solution, allowing a Python-based application like Home Assistant, running on a Linux device such as a Raspberry Pi, to communicate directly with the Seneye hardware.

The user's prior failure with an MQTT-based system was likely an attempt to bridge data from either the Cloud API or the LDE. The unreliability experienced stems from the fundamental architectural weakness of relying on a network stack (whether local or internet-facing) for a task that can be accomplished via a direct, physical connection. By choosing the direct USB path enabled by pyseneye, the architecture presented in this report makes a deliberate engineering decision to engineer for maximum resilience, directly addressing the root cause of the user's previous failure. This principle—that a direct physical connection is superior to a networked abstraction for this critical monitoring use case—is the central thesis of this report.

## Section 2: Architecting the Communication Layer: From Raw USB to Actionable Data

With the foundational concepts established, the next step is to construct the software layer that bridges the Home Assistant application and the physical Seneye hardware. This involves descending through layers of abstraction, from the low-level intricacies of raw USB communication to the clean, high-level API provided by the pyseneye library. A thorough understanding of this communication stack is essential for both successful implementation and effective troubleshooting.

### 2.1 A Technical Deep-Dive into PyUSB

At the lowest level of our Python application stack is PyUSB, a powerful library that provides direct access to the host machine's USB subsystem.17 It serves as a wrapper around underlying system libraries like

libusb, abstracting away the OS-specific details of USB communication.

* **Device Discovery:** The first step in any USB interaction is to locate the target device. USB devices are identified by a unique combination of a Vendor ID (VID) and a Product ID (PID), which are assigned to manufacturers by the USB Implementers Forum. PyUSB provides the usb.core.find() function for this purpose. A simple script can iterate through all connected devices or find a specific one by its VID and PID.17 The Seneye SUD can be located by querying for its specific identifiers.
* **The USB Device Hierarchy:** Communication with a USB device is not monolithic; it follows a structured hierarchy that must be navigated to access the correct data channel.20
  1. **Device:** The physical hardware connected to the USB port.
  2. **Configuration:** A device can have one or more configurations, which define its power consumption and the set of interfaces it offers. Most devices, including the Seneye SUD, have only one configuration. dev.set\_configuration() is used to activate it.
  3. **Interface:** A configuration is composed of one or more interfaces, which represent a functional grouping. For example, a multifunction printer might have one interface for printing and another for scanning.
  4. **Endpoint:** Each interface has one or more endpoints, which are the actual unidirectional channels for data transfer. Endpoints are designated as either IN (device-to-host) or OUT (host-to-device). They also have a transfer type, such as Bulk, Interrupt, or Isochronous, which defines the data transfer characteristics.20 The Seneye device uses Interrupt transfers for its HID-based communication.24
* **Managing Kernel Drivers:** On modern operating systems like Linux (which forms the basis of Raspberry Pi OS and Home Assistant OS), the kernel will often automatically detect a new USB device and attach a generic driver to it (e.g., usbhid for Human Interface Devices). This kernel driver "claims" the device's interface, preventing user-space applications like our Python script from accessing it directly.20 This will result in a "Resource busy" error. To resolve this,  
  PyUSB provides the dev.detach\_kernel\_driver() method. It is a critical and mandatory step to tell the kernel to release its claim on the interface, allowing our application to take exclusive control with usb.util.claim\_interface(). After the application is finished, it is good practice to re-attach the driver using dev.attach\_kernel\_driver().
* **Endpoint Communication:** Once an interface is claimed, data can be transferred using the dev.read() and dev.write() methods. These functions target a specific endpoint address and transfer a raw array of bytes.26 The  
  write() command sends a specific sequence of bytes that the device's firmware understands as a command, and the read() command listens on an IN endpoint for the device's response. It is crucial to include a timeout parameter in the read() call to prevent the program from blocking indefinitely if the device does not respond.25 This low-level interaction requires precise knowledge of the device's proprietary command set and response format.

### 2.2 The pyseneye Library: A High-Level Abstraction for the SUD

While it is possible to interact with the Seneye device using PyUSB directly, it would require reverse-engineering the command protocol from the SUDDriver examples and manually parsing the binary response data. This is a complex, error-prone process. The pyseneye library serves as an expertly crafted abstraction layer that encapsulates this complexity, providing a simple, high-level, and Pythonic API for device interaction.11

* **Core Class: SUDevice:** The library exposes a central class, pyseneye.sud.SUDevice, which represents the physical sensor device. Instantiating this class automatically handles the underlying PyUSB device discovery, configuration, and kernel driver detachment.11
* **The Action Enum:** Instead of requiring the developer to know the raw byte commands, pyseneye defines an Action enumeration that lists the available high-level commands. These include ENTER\_INTERACTIVE\_MODE to initialize a session with the device, and most importantly, SENSOR\_READING to request a full data readout.11
* **The action() Method:** The complexity of writing a command and reading the response is simplified into a single method call: response = device.action(Action.SENSOR\_READING). This one line of code transparently handles the underlying dev.write() and dev.read() calls, including any necessary handshaking and validation.
* **Response Parsing:** The true value of the library is demonstrated in its response handling. The raw byte array returned by the device is not passed back to the user. Instead, it is parsed into a structured SensorReadingResponse object.29 This object exposes the sensor data through clean, clearly named properties. For example, after a successful  
  SENSOR\_READING action, the pH can be accessed via response.ph, the temperature via response.temperature, and, critically for this project's requirements, the advanced light metrics via response.par, response.pur, and response.lux.29 The library handles all the necessary byte-level manipulation, such as unpacking binary data structures and applying the correct floating-point conversions.

The relationship between these software layers forms a communication stack: the Home Assistant integration will make high-level calls to pyseneye, which in turn translates these into a series of low-level PyUSB operations, which then interface with the system's libusb library to communicate with the kernel and, ultimately, the hardware. This layered abstraction is powerful, but it also means that a potential issue could reside at any level. An error such as "Device not found" could be a physical connection problem, a Linux permissions issue preventing libusb access, a PyUSB backend configuration error, or a bug in pyseneye's discovery logic. By understanding this stack, one can troubleshoot methodically, isolating the problem layer by layer, which is a far more effective approach than random guesswork.

## Section 3: Building a Production-Grade Home Assistant Custom Integration

This section details the complete process of constructing a modern, robust, and maintainable custom integration for Home Assistant. The objective is to move beyond a simple proof-of-concept and engineer a component that adheres to current best practices, ensuring stability, efficiency, and future compatibility. The architecture will leverage the pyseneye library for device communication and will be built around Home Assistant's DataUpdateCoordinator pattern for optimal data management.

### 3.1 Anatomy of a Home Assistant Custom Component

A Home Assistant custom integration is a collection of Python files and a manifest file organized within a specific directory structure. This structure allows Home Assistant's core to discover, load, and manage the integration. For this project, the component will reside in <config>/custom\_components/seneye/, where <config> is the main Home Assistant configuration directory.30

The essential files for this integration are:

* manifest.json: This file serves as the integration's metadata descriptor. It tells Home Assistant the component's name, version, documentation URL, and, most importantly, its Python package dependencies.30
* \_\_init\_\_.py: This is the primary entry point for the integration. It contains the core setup and teardown logic. In a modern integration, its main responsibility is to initialize and configure the data coordinator and register services.30
* sensor.py: This file is responsible for defining the sensor platforms. It contains the code that creates the individual sensor entities (e.g., for temperature, pH, PAR) that will be visible in the Home Assistant user interface.30
* const.py: While not strictly required, it is a strongly recommended best practice to store all constants, such as the integration's domain (DOMAIN = "seneye"), in a separate file. This improves code readability and maintainability.
* config\_flow.py: This file enables UI-based configuration, allowing users to add and set up the integration from the Home Assistant frontend without editing YAML files.
* services.yaml: This file defines the custom services the integration provides, making them discoverable and usable within Home Assistant's automation and script editors.

### 3.2 manifest.json: The Integration's Identity Card

The manifest.json file is the first file Home Assistant reads when loading the integration. It must contain accurate information for the component to be loaded correctly.

JSON

{  
 "domain": "seneye",  
 "name": "Seneye USB Sensor",  
 "documentation": "https://github.com/mcclown/pyseneye",  
 "issue\_tracker": "https://github.com/mcclown/pyseneye/issues",  
 "codeowners":,  
 "requirements": [  
 "pyseneye==0.1.6"  
 ],  
 "config\_flow": true,  
 "iot\_class": "local\_polling",  
 "version": "1.1.0"  
}

* **domain**: A unique, lowercase identifier for the integration. This must match the directory name (seneye).30
* **name**: A user-friendly name that will be displayed in the Home Assistant UI.
* **requirements**: This is a critical entry. It is an array of Python packages that Home Assistant's package manager will automatically install upon loading the integration. By specifying "pyseneye==0.1.6", we ensure that the necessary communication library is available without requiring manual installation by the end-user.35
* **config\_flow**: Setting this to true is essential. It informs Home Assistant that the integration uses a UI-based setup process defined in config\_flow.py, enabling it to be added from the integrations page.
* **iot\_class**: This key helps Home Assistant classify the integration's behavior. "local\_polling" accurately describes our architecture: the integration runs locally and actively polls the device for data.35
* **version**: For custom components, a version number following Semantic Versioning is required.

### 3.3 **init**.py and the Coordinator Pattern: Orchestrating Data Flow

The \_\_init\_\_.py file orchestrates the setup of the integration. The modern approach for polled devices is the DataUpdateCoordinator pattern, which provides a centralized and efficient mechanism for fetching data.31 This pattern ensures that the physical USB device is queried only once per update interval, and the resulting data is then distributed to all associated sensor entities. This file is also responsible for registering any custom services the integration provides.

Python

# custom\_components/seneye/\_\_init\_\_.py  
  
import logging  
from datetime import timedelta  
from pyseneye.sud import SUDevice, Action, DeviceType  
  
from homeassistant.core import HomeAssistant, ServiceCall  
from homeassistant.config\_entries import ConfigEntry  
from homeassistant.helpers.update\_coordinator import DataUpdateCoordinator, UpdateFailed  
  
from.const import DOMAIN  
  
\_LOGGER = logging.getLogger(\_\_name\_\_)  
PLATFORMS = ["sensor"]  
SERVICE\_FORCE\_UPDATE = "force\_update"  
  
async def async\_setup\_entry(hass: HomeAssistant, entry: ConfigEntry) -> bool:  
 """Set up Seneye from a config entry."""  
   
 coordinator = SeneyeDataUpdateCoordinator(hass)  
 await coordinator.async\_config\_entry\_first\_refresh()  
  
 hass.data.setdefault(DOMAIN, {})  
 hass.data[entry.entry\_id] = coordinator  
  
 await hass.config\_entries.async\_forward\_entry\_setups(entry, PLATFORMS)  
  
 # Register the custom service  
 async def async\_force\_update\_service(call: ServiceCall) -> None:  
 """Handle the service call to force a refresh."""  
 \_LOGGER.info("Service 'seneye.force\_update' called: forcing a data refresh")  
 await coordinator.async\_request\_refresh()  
  
 hass.services.async\_register(  
 DOMAIN, SERVICE\_FORCE\_UPDATE, async\_force\_update\_service  
 )  
  
 return True  
  
async def async\_unload\_entry(hass: HomeAssistant, entry: ConfigEntry) -> bool:  
 """Unload a config entry."""  
 unload\_ok = await hass.config\_entries.async\_unload\_platforms(entry, PLATFORMS)  
 if unload\_ok:  
 hass.data.pop(entry.entry\_id)  
 # Unregister the service when the integration is unloaded  
 hass.services.async\_remove(DOMAIN, SERVICE\_FORCE\_UPDATE)  
  
 return unload\_ok  
  
class SeneyeDataUpdateCoordinator(DataUpdateCoordinator):  
 """Class to manage fetching Seneye data."""  
  
 def \_\_init\_\_(self, hass: HomeAssistant):  
 """Initialize the coordinator."""  
 super().\_\_init\_\_(  
 hass,  
 \_LOGGER,  
 name=DOMAIN,  
 update\_interval=timedelta(minutes=30),  
 )  
  
 async def \_async\_update\_data(self):  
 """Fetch data from the Seneye device."""  
 try:  
 return await self.hass.async\_add\_executor\_job(self.\_get\_seneye\_data)  
 except Exception as err:  
 raise UpdateFailed(f"Error communicating with Seneye device: {err}")  
  
 def \_get\_seneye\_data(self):  
 """Synchronous function to communicate with the USB device."""  
 device = SUDevice()  
 try:  
 # Enter interactive mode to prepare for reading  
 device.action(Action.ENTER\_INTERACTIVE\_MODE)  
 # Request a full sensor reading  
 response = device.action(Action.SENSOR\_READING)  
 return response  
 finally:  
 # Always ensure the device connection is closed  
 device.close()

This updated code now includes the registration of the force\_update service. When seneye.force\_update is called from an automation or the developer tools, the async\_force\_update\_service function is executed. This function then calls coordinator.async\_request\_refresh(), which is the standard Home Assistant method to trigger an immediate, on-demand poll of the device data, bypassing the regular 30-minute interval.

### 3.4 sensor.py: The Heart of the Integration

This file defines the sensor entities that will be presented to the user. By leveraging the DataUpdateCoordinator and the SensorEntityDescription dataclass, we can create a clean, efficient, and easily extensible implementation. The SensorEntityDescription allows us to define the static characteristics of each sensor (name, icon, unit, etc.) in a declarative manner, and then use a single generic sensor class to bring them all to life.

First, a specification table defines the entities to be created. This serves as a blueprint for the implementation and confirms that all user-requested metrics are included.

| Measurement | Entity ID Suffix | Friendly Name | Unit of Measurement | Device Class | Icon |
| --- | --- | --- | --- | --- | --- |
| Temperature | \_temperature | Seneye Temperature | °C | temperature | mdi:thermometer |
| pH | \_ph | Seneye pH | pH | None | mdi:ph |
| NH₃ | \_nh3 | Seneye NH₃ | ppm | None | mdi:molecule |
| In Water | \_in\_water | Seneye In Water | None | None | mdi:water-check |
| Slide Status | \_slide\_status | Seneye Slide Status | None | None | mdi:wiper |
| **PAR** | \_par | Seneye PAR | μmol/m²/s | None | mdi:solar-power |
| **PUR** | \_pur | Seneye PUR | % | None | mdi:theme-light-dark |
| **LUX** | \_lux | Seneye LUX | lx | illuminance | mdi:brightness-5 |
| Kelvin | \_kelvin | Seneye Kelvin | K | None | mdi:temperature-kelvin |

The following code implements these sensors.

Python

# custom\_components/seneye/sensor.py  
  
from \_\_future\_\_ import annotations  
import logging  
from dataclasses import dataclass  
  
from homeassistant.components.sensor import (  
 SensorEntity,  
 SensorEntityDescription,  
 SensorStateClass,  
)  
from homeassistant.const import UnitOfTemperature, CONCENTRATION\_PARTS\_PER\_MILLION, LIGHT\_LUX  
from homeassistant.core import HomeAssistant  
from homeassistant.config\_entries import ConfigEntry  
from homeassistant.helpers.entity\_platform import AddEntitiesCallback  
from homeassistant.helpers.update\_coordinator import CoordinatorEntity  
  
from.const import DOMAIN  
from. import SeneyeDataUpdateCoordinator  
  
\_LOGGER = logging.getLogger(\_\_name\_\_)  
  
@dataclass  
class SeneyeSensorEntityDescription(SensorEntityDescription):  
 """Describes a Seneye sensor entity."""  
 value\_fn: callable | None = None  
  
SENSORS: tuple = (  
 SeneyeSensorEntityDescription(  
 key="temperature",  
 name="Temperature",  
 native\_unit\_of\_measurement=UnitOfTemperature.CELSIUS,  
 device\_class="temperature",  
 state\_class=SensorStateClass.MEASUREMENT,  
 value\_fn=lambda data: data.temperature,  
 ),  
 SeneyeSensorEntityDescription(  
 key="ph",  
 name="pH",  
 native\_unit\_of\_measurement="pH",  
 icon="mdi:ph",  
 state\_class=SensorStateClass.MEASUREMENT,  
 value\_fn=lambda data: data.ph,  
 ),  
 SeneyeSensorEntityDescription(  
 key="nh3",  
 name="NH3",  
 native\_unit\_of\_measurement=CONCENTRATION\_PARTS\_PER\_MILLION,  
 icon="mdi:molecule",  
 state\_class=SensorStateClass.MEASUREMENT,  
 value\_fn=lambda data: data.nh3,  
 ),  
 SeneyeSensorEntityDescription(  
 key="in\_water",  
 name="In Water",  
 icon="mdi:water-check",  
 value\_fn=lambda data: "In Water" if data.in\_water else "Out of Water",  
 ),  
 SeneyeSensorEntityDescription(  
 key="slide\_ok",  
 name="Slide Status",  
 icon="mdi:wiper",  
 value\_fn=lambda data: "OK" if not data.slide\_expired else "Expired",  
 ),  
 SeneyeSensorEntityDescription(  
 key="par",  
 name="PAR",  
 native\_unit\_of\_measurement="μmol/m²/s",  
 icon="mdi:solar-power",  
 state\_class=SensorStateClass.MEASUREMENT,  
 value\_fn=lambda data: data.par,  
 ),  
 SeneyeSensorEntityDescription(  
 key="pur",  
 name="PUR",  
 native\_unit\_of\_measurement="%",  
 icon="mdi:theme-light-dark",  
 state\_class=SensorStateClass.MEASUREMENT,  
 value\_fn=lambda data: data.pur,  
 ),  
 SeneyeSensorEntityDescription(  
 key="lux",  
 name="LUX",  
 native\_unit\_of\_measurement=LIGHT\_LUX,  
 device\_class="illuminance",  
 state\_class=SensorStateClass.MEASUREMENT,  
 value\_fn=lambda data: data.lux,  
 ),  
 SeneyeSensorEntityDescription(  
 key="kelvin",  
 name="Kelvin",  
 native\_unit\_of\_measurement="K",  
 icon="mdi:temperature-kelvin",  
 state\_class=SensorStateClass.MEASUREMENT,  
 value\_fn=lambda data: data.kelvin,  
 ),  
)  
  
async def async\_setup\_entry(  
 hass: HomeAssistant,  
 entry: ConfigEntry,  
 async\_add\_entities: AddEntitiesCallback,  
) -> None:  
 """Set up the Seneye sensor platform."""  
 coordinator: SeneyeDataUpdateCoordinator = hass.data[entry.entry\_id]  
   
 entities =  
 async\_add\_entities(entities)  
  
class SeneyeSensor(CoordinatorEntity, SensorEntity):  
 """Representation of a Seneye sensor."""  
  
 entity\_description: SeneyeSensorEntityDescription  
  
 def \_\_init\_\_(  
 self,  
 coordinator: SeneyeDataUpdateCoordinator,  
 description: SeneyeSensorEntityDescription,  
 ) -> None:  
 """Initialize the sensor."""  
 super().\_\_init\_\_(coordinator)  
 self.entity\_description = description  
 self.\_attr\_unique\_id = f"{coordinator.config\_entry.entry\_id}\_{description.key}"  
 self.\_attr\_name = f"Seneye {description.name}"  
  
 @property  
 def native\_value(self) -> str | int | float | None:  
 """Return the state of the sensor."""  
 if self.coordinator.data and self.entity\_description.value\_fn:  
 return self.entity\_description.value\_fn(self.coordinator.data)  
 return None

This implementation creates a scalable and maintainable sensor platform. To add a new sensor in the future, one would only need to add a new SeneyeSensorEntityDescription to the SENSORS tuple; no other code changes would be required.36 The

SeneyeSensor class inherits from CoordinatorEntity, which automatically handles subscribing to updates from the coordinator, ensuring that the sensor's state is refreshed whenever new data is fetched from the device.

### 3.5 config\_flow.py: Enabling UI-Based Setup

To eliminate the need for manual YAML configuration, we introduce a config\_flow.py file. Since the Seneye device is connected directly via USB and requires no user-configurable parameters (like an IP address or API key), this will be a "zero-configuration" flow. Its primary job is to ensure that only one instance of the integration is configured and then create the configuration entry.

Python

# custom\_components/seneye/config\_flow.py  
  
from \_\_future\_\_ import annotations  
  
from homeassistant import config\_entries  
from homeassistant.data\_entry\_flow import FlowResult  
  
from.const import DOMAIN  
  
class SeneyeConfigFlow(config\_entries.ConfigFlow, domain=DOMAIN):  
 """Handle a config flow for Seneye USB."""  
  
 VERSION = 1  
  
 async def async\_step\_user(  
 self, user\_input: dict | None = None  
 ) -> FlowResult:  
 """Handle the initial step."""  
 # Only allow a single instance of the integration to be configured.  
 if self.\_async\_current\_entries():  
 return self.async\_abort(reason="single\_instance\_allowed")  
  
 # If the user initiates the flow, create the config entry.  
 # No user input is needed as the device is auto-detected locally.  
 if user\_input is None:  
 return self.async\_show\_form(step\_id="user")  
  
 return self.async\_create\_entry(title="Seneye USB Monitor", data={})

This code defines a simple user-initiated flow. When a user adds the "Seneye USB Sensor" integration from the UI, this flow first checks if an entry already exists. If so, it aborts. If not, it presents a confirmation step and then creates the entry, which triggers the async\_setup\_entry function in \_\_init\_\_.py.

### 3.6 services.yaml: Defining Custom Services

To make the force\_update service discoverable and user-friendly within Home Assistant (e.g., in the Developer Tools > Services UI), we must define it in a services.yaml file. This file provides metadata about the service, such as its name and description.

YAML

# custom\_components/seneye/services.yaml  
  
force\_update:  
 name: Force Update  
 description: "Forces an immediate update of all Seneye sensor data from the USB device."

This simple file makes the service easy to find and understand for users who want to build automations or manually trigger a data refresh.

## Section 4: Deployment, Validation, and Advanced Usage

With the custom component fully architected and coded, this final section provides a comprehensive guide to deploying, configuring, and troubleshooting the integration in a live Home Assistant environment. Proper deployment, particularly on a Linux-based system like a Raspberry Pi, requires attention to system-level prerequisites and permissions.

### 4.1 Installation and Configuration Guide

* **Prerequisites:** The pyseneye library, and by extension PyUSB, relies on the libusb system library to interface with the operating system's USB controller. This library must be installed on the host system. For Debian-based systems, including Raspberry Pi OS, this can be accomplished with a single command 17:  
  Bash  
  sudo apt-get update && sudo apt-get install libusb-1.0-0
* **Component Deployment:** The custom component files must be placed in the correct directory for Home Assistant to discover them.
  1. Access the Home Assistant configuration directory. This is typically /config in Home Assistant OS or a mounted volume in a container installation.34
  2. Inside this directory, create a new folder named custom\_components if it does not already exist.
  3. Inside custom\_components, create a folder named seneye.
  4. Copy all the integration files (\_\_init\_\_.py, sensor.py, manifest.json, const.py, config\_flow.py, and services.yaml) into the seneye directory.31
* **Restart and Configuration:**
  1. After copying the files, restart the Home Assistant server.
  2. Once restarted, navigate to **Settings > Devices & Services**.
  3. Click the **+ ADD INTEGRATION** button in the bottom right corner.
  4. Search for "Seneye USB Sensor" and select it.
  5. Follow the on-screen confirmation to add the integration. The sensors should now appear under the Seneye device.

### 4.2 Troubleshooting and Diagnostics

Direct hardware interaction can present unique challenges, most of which are related to permissions and device access.

* **USB Permissions on Linux:** This is the most common failure point. By default, standard users (including the user that the Home Assistant process runs as) do not have direct write access to USB hardware. Attempting to run the integration without correct permissions will result in an [Errno 13] Access denied error in the logs.21 The proper and permanent solution is to create a  
  udev rule. udev is the Linux subsystem for managing device events and permissions.
  1. Create a new file at /etc/udev/rules.d/99-seneye.rules.
  2. Add the following line to the file. This rule identifies the Seneye device by its Vendor and Product ID and grants read/write permissions to all users (MODE="0666").  
     SUBSYSTEM=="usb", ATTR{idVendor}=="2437", ATTR{idProduct}=="0100", MODE="0666"
  3. Reload the udev rules and trigger a re-scan by running:  
     Bash  
     sudo udevadm control --reload-rules && sudo udevadm trigger
  4. Unplug and replug the Seneye device to ensure the new rule is applied.
* **Device not found Errors:** If the integration reports that the device cannot be found, follow this diagnostic checklist 25:
  1. **Physical Connection:** Verify the USB cable is securely connected to both the Seneye device and the host machine.
  2. **OS-Level Detection:** Open a terminal on the host machine and run the lsusb command. The output should include a line for the Seneye device, confirming that the operating system itself can see the hardware. The entry should show ID 2437:0100.
  3. **Home Assistant Logs:** Check the Home Assistant logs (**Settings > System > Logs**) for any errors from PyUSB or the custom\_components.seneye logger. This may indicate a backend issue or a more specific error message.
* **Logging:** To get more detailed information during troubleshooting, enable debug logging for the custom component. Add the following to your configuration.yaml and restart Home Assistant 38:  
  YAML  
  logger:  
   default: info  
   logs:  
   custom\_components.seneye: debug  
    
  This will provide verbose output from the integration during its setup phase and each time the coordinator polls the device for data.

### 4.3 Using the force\_update Service

The newly added seneye.force\_update service can be used in various ways:

* **Developer Tools:** Navigate to **Developer Tools > Services**. Select Seneye USB Sensor: Force Update from the service dropdown and click "Call Service". Check the logs to confirm that a data refresh was initiated.
* **Automations:** Create an automation that calls this service. For example, you could create a button in your dashboard that, when pressed, calls the seneye.force\_update service to get the latest readings on demand.
* **Scripts:** Use the service within scripts to chain multiple actions together, such as forcing an update before sending a notification with the latest sensor values.

## Conclusion

This report has detailed a comprehensive, expert-level architecture for the local-first integration of Seneye USB sensors into Home Assistant. By methodically rejecting network- and cloud-based data pathways in favor of a direct USB communication strategy, this solution directly addresses the critical requirement for system reliability and data integrity. The implementation leverages the community-developed pyseneye library, which provides a high-level abstraction over the complexities of the underlying USB protocol.

The custom component itself is engineered using modern Home Assistant development patterns, including the DataUpdateCoordinator for efficient, centralized polling, a UI-based ConfigFlow for simple setup, and a custom service for on-demand updates. This approach not only delivers a robust and performant integration but also ensures future maintainability and an excellent user experience.

The final solution successfully fulfills all project requirements: it provides a stable alternative to unreliable network-based methods, it integrates the full suite of advanced lighting metrics including PAR, PUR, and LUX, and it creates a powerful, local, and extensible platform for sophisticated environmental monitoring. This architecture stands as a blueprint for integrating critical hardware into a smart home ecosystem where reliability is not a feature, but a foundational necessity.

#### Works cited

1. seneye USB Device (SUD) Driver Developer Information - GitHub, accessed August 19, 2025, <https://github.com/seneye/SUDDriver>
2. Photosynthetically active radiation - Wikipedia, accessed August 19, 2025, <https://en.wikipedia.org/wiki/Photosynthetically_active_radiation>
3. How to read PAR tables - The 2Hr Aquarist, accessed August 19, 2025, <https://www.2hraquarist.com/blogs/light-3pillars/par_for_planted_tank>
4. What is PAR & PUR - Seneye Answer, accessed August 19, 2025, <https://answers.seneye.com/en/Aquarium_help/what_is_PAR_%3F>
5. Lighting the Aquarium Part 7-Photosynthetically Usable Radiation, accessed August 19, 2025, <https://orphek.com/lighting-the-aquarium-part-7-photosynthetically-usable-radiation/>
6. PUR Photosynthetically Useable Radiation - Orphek Reef Aquarium LED Lighting, accessed August 19, 2025, <https://orphek.com/pur-photosynthetically-useable-radiation/>
7. answers.seneye.com, accessed August 19, 2025, <https://answers.seneye.com/en/Aquarium_help/what_is_PAR_%3F#:~:text=The%20part%20of%20the%20PAR,PUR%20in%20a%20PAR%20value.>
8. What is PAR & PUR - Seneye Answer, accessed August 19, 2025, [https://answers.seneye.com/index.php?title=en/Aquarium\_help/What\_is\_PAR\_%26\_PUR\_%3F](https://answers.seneye.com/index.php?title=en/Aquarium_help/What_is_PAR_%26_PUR_?)
9. What is Photosynthetically Active Radiation (PAR)? - Monnit Knowledge Base, accessed August 19, 2025, <https://support.monnit.com/article/82-what-is-photosynthetically-active-radiation>
10. seneye/LDE: Seneye Local Data Exchange - GitHub, accessed August 19, 2025, <https://github.com/seneye/LDE>
11. Adding support for Seneye Aquarium & Pond Sensors (removing the need for a Seneye Web Server) - Home Assistant Community, accessed August 19, 2025, <https://community.home-assistant.io/t/adding-support-for-seneye-aquarium-pond-sensors-removing-the-need-for-a-seneye-web-server/101667>
12. Adding support for Seneye Aquarium & Pond Sensors (removing the need for a Seneye Web Server) - #17 by DirkTas67 - Third party integrations - Home Assistant Community, accessed August 19, 2025, <https://community.home-assistant.io/t/adding-support-for-seneye-aquarium-pond-sensors-removing-the-need-for-a-seneye-web-server/101667/17>
13. Adding support for Seneye Aquarium & Pond Sensors (removing the need for a Seneye Web Server) - Page 2 - Home Assistant Community, accessed August 19, 2025, <https://community.home-assistant.io/t/adding-support-for-seneye-aquarium-pond-sensors-removing-the-need-for-a-seneye-web-server/101667?page=2>
14. Seneye hobbyist developer information, accessed August 19, 2025, <https://answers.seneye.com/en/Seneye_Products/Seneye_hobbyist_developer_information>
15. Adding support for Seneye Aquarium & Pond Sensors (removing the need for a Seneye Web Server) - #92 by notovich - Third party integrations - Home Assistant Community, accessed August 19, 2025, <https://community.home-assistant.io/t/adding-support-for-seneye-aquarium-pond-sensors-removing-the-need-for-a-seneye-web-server/101667/92>
16. Adding support for Seneye Aquarium & Pond Sensors (removing the need for a Seneye Web Server) - Page 3 - Third party integrations - Home Assistant Community, accessed August 19, 2025, <https://community.home-assistant.io/t/adding-support-for-seneye-aquarium-pond-sensors-removing-the-need-for-a-seneye-web-server/101667?page=3>
17. pyusb/pyusb: Easy USB access for Python - GitHub, accessed August 19, 2025, <https://github.com/pyusb/pyusb>
18. PyUSB - Easy USB access on Python - GitHub Pages, accessed August 19, 2025, <https://pyusb.github.io/pyusb/>
19. Use PyUSB to Find Vendor and Product IDs for USB Devices | Web Design Greenville SC, accessed August 19, 2025, <https://www.orangecoat.com/how-to/use-pyusb-to-find-vendor-and-product-ids-for-usb-devices>
20. USB Communication with Python and PyUSB - Martin Vidner, accessed August 19, 2025, <http://mvidner.blogspot.com/2017/01/usb-communication-with-python-and-pyusb.html>
21. Read and Decode Data From Your Mouse Using This PyUSB Hack ..., accessed August 19, 2025, <https://www.orangecoat.com/how-to/read-and-decode-data-from-your-mouse-using-this-pyusb-hack>
22. How to control a USB port using Python, accessed August 19, 2025, <https://www.pythonhelp.org/tutorials/how-to-control-usb-port/>
23. RaspberryPi PICO and PYUSB: Unlocking the Power of Custom USB Endpoints, accessed August 19, 2025, <https://www.shekharverma.com/python-raspberrypi-pico-and-pyusb-unlocking-the-power-of-custom-usb-endpoints/>
24. Pyusb freeze when using Dev.read(nr,nr,timeout=1000) - Python Forum, accessed August 19, 2025, <https://python-forum.io/thread-1526.html>
25. How can I stop pyusb from reading if nothing comes from my hid device - Reddit, accessed August 19, 2025, <https://www.reddit.com/r/learnpython/comments/5jdp1c/how_can_i_stop_pyusb_from_reading_if_nothing/>
26. Using Python and the LIBUSB Library with USB Devices in Linux and Windows based systems., accessed August 19, 2025, <https://www.ontrak.net/LibUSBPy.htm>
27. Reading from USB mouse on Windows · Issue #334 - GitHub, accessed August 19, 2025, <https://github.com/pyusb/pyusb/issues/334>
28. pyUSB get a continuous stream of data from sensor - Stack Overflow, accessed August 19, 2025, <https://stackoverflow.com/questions/29291631/pyusb-get-a-continuous-stream-of-data-from-sensor>
29. pyseneye 0.0.1 documentation - Read the Docs, accessed August 19, 2025, <https://pyseneye.readthedocs.io/en/latest/pyseneye.html>
30. Creating your first integration | Home Assistant Developer Docs, accessed August 19, 2025, <https://developers.home-assistant.io/docs/creating_component_index/>
31. Integration file structure | Home Assistant Developer Docs, accessed August 19, 2025, <https://developers.home-assistant.io/docs/creating_integration_file_structure/>
32. Creating a custom component for home assistant - Blog - TheStaticTurtle, accessed August 19, 2025, <https://blog.thestaticturtle.fr/creating-a-custom-component-for-homeassistant/>
33. Creating your first integration - Home Assistant Developer Docs, accessed August 19, 2025, <https://developers.home-assistant.io/docs/creating_component_index>
34. Configuration.yaml - Home Assistant, accessed August 19, 2025, <https://www.home-assistant.io/docs/configuration/>
35. Building a Home Assistant Custom Component Part 1: Project ..., accessed August 19, 2025, <https://aarongodfrey.dev/home%20automation/building_a_home_assistant_custom_component_part_1/>
36. Custom Component - How to create multiple sensors? - Home Assistant Community, accessed August 19, 2025, <https://community.home-assistant.io/t/custom-component-how-to-create-multiple-sensors/48796>
37. python - PyUSB Can't find device - Stack Overflow, accessed August 19, 2025, <https://stackoverflow.com/questions/8555930/pyusb-cant-find-device>
38. Creating a Home Assistant component - Stephan van Rooij, accessed August 19, 2025, <https://svrooij.io/2023/01/18/home-assistant-component/>