A Definitive Guide to a Resilient Seneye USB Bridge for Home Assistant

Executive Summary: A Blueprint for Stability

The journey to a seamlessly integrated smart aquarium can be fraught with technical challenges, leading to frustration when seemingly simple components fail to communicate reliably. Past experiences with an MQTT-based bridge for the Seneye Reef device have highlighted the fragility of solutions that rely on multiple, loosely-coupled software layers. Intermittent failures, data corruption, and complete service outages are often symptoms of a fundamentally flawed architecture rather than isolated bugs. This report outlines a new, robust, and resilient approach that replaces the previous system with a more direct and stable foundation.

This new architecture is built upon the USB/IP protocol, a mature and powerful technology integrated directly into the Linux kernel. Instead of relying on a high-level script to interpret and relay data, this method operates at a much lower level, effectively creating a wireless USB extension cord. This approach fundamentally changes the system's dynamics, increasing stability by orders of magnitude. The architecture rests on three core pillars:

\* The USB/IP Server: A dedicated Raspberry Pi Zero 2W will act as a network-attached USB port. Its sole function is to capture the raw data from the physically connected Seneye device and broadcast it over the local network. A custom, auto-recovering service will ensure this connection is re-established automatically after power cycles, network interruptions, or device lockups.

\* The USB/IP Client: A specialized Home Assistant Add-on will be installed. This client listens for the broadcast from the Raspberry Pi and creates a virtual USB port within the Home Assistant Operating System. To Home Assistant, the remote Seneye device will appear as if it were physically plugged directly into the machine running the home automation platform.

\* The Seneye Integration: With the virtual connection established, a community-developed Home Assistant custom component, pyseneye, will be used to communicate with this new "local" device. This component understands the Seneye-specific protocol and translates the raw USB data into readable sensor entities for temperature, pH, and ammonia.

By following this guide, a system will be constructed that is not only functional but is architected for resilience. Each component is designed to handle failure gracefully and recover automatically, transforming the Seneye monitor from a source of frustration into a reliable cornerstone of a fully automated reef tank ecosystem.

Part 1: Deconstructing the Past, Architecting the Future

To build a truly stable system, it is essential to first understand the failure points of the previous approach. The frustrations experienced with the MQTT bridge were not random; they were the predictable result of an architecture ill-suited to the task. By dissecting this past failure, the inherent superiority of the new USB/IP paradigm becomes clear.

1.1 Autopsy of the MQTT Bridge: A Tale of Compounding Instabilities

The previous attempt involved a Raspberry Pi running a script to read the Seneye device, publish the readings to an MQTT broker, which Home Assistant would then subscribe to. The user described a recurring failure mode: the temperature probe reading would "creep out of alignment," followed by a total cessation of all reported services [User Query]. This observation is the key to understanding the system's collapse.

The "temperature creep" was almost certainly not a physical malfunction of the temperature probe itself. Instead, it was the first visible symptom of a deeper issue: a communication failure at the USB level. The Seneye USB Device (SUD) is known to be sensitive. If its specific communication protocol is interrupted—for example, by missing a final "BYESUD" command—the device can enter a "timeout-locked state". In this state, it may stop responding correctly or begin sending corrupted data until it is physically power-cycled by unplugging and replugging it.

The Python script acting as the bridge was likely designed to parse a specific data structure from the device. When the Seneye entered a locked state and began sending malformed data, the script would encounter a value it could not interpret. This would lead to a parsing error, causing the script to crash. The crash of the script would instantly sever the connection to the MQTT broker. Consequently, no new data would be published, and all the Seneye entities in Home Assistant would go stale, appearing to have stopped reporting simultaneously. This chain of events perfectly matches the described failure.

The root cause, therefore, was not a single bug but a cascade of failures originating from an architectural weakness. The system was a fragile chain of dependencies:

\* Unreliable Hardware: The Seneye device itself is prone to communication stalls.

\* Brittle Software: The user-space script was not robust enough to handle the hardware's error states.

\* Coupled Responsibilities: The script was tasked with both managing the low-level USB communication and interpreting the high-level sensor data. A failure in one task caused the entire process to fail.

This analysis validates the user's frustration. The problem was not a simple coding error but a design that was intolerant to the inevitable imperfections of the hardware it was meant to control.

1.2 The USB/IP Paradigm: A More Perfect (and Simpler) Union

The USB/IP protocol offers a fundamentally different and more robust solution by operating at a lower, more direct level. Its purpose is to encapsulate raw "USB I/O messages" into TCP/IP packets and transmit them across a network. This creates a transparent bridge where a client machine can interact with a remote USB device as if it were physically connected.

The architecture is a classic client-server model. The "server" is the computer with the physical USB device attached—in this case, the Raspberry Pi. It runs a kernel module (usbip\_host) that makes the device available on the network. The "client" is the computer that wishes to use the device—in this case, the machine running Home Assistant. It runs a corresponding kernel module called vhci-hcd (Virtual Host Controller Interface), which creates a virtual USB hub that the remote device can be "plugged into". This technology is not experimental; it has been a standard part of the mainline Linux kernel for many years, ensuring a high degree of stability and performance.

The most significant advantage of this approach is the decoupling of responsibilities. The previous MQTT bridge was responsible for both communicating with the device and interpreting its data. The USB/IP architecture cleanly separates these concerns:

\* The Raspberry Pi's Role (Server): Its only job is to maintain the USB connection and forward the raw, unmodified USB packets over the network. It does not need to know what a "temperature reading" is or how to parse it. It is, in effect, a "dumb" network adapter for the USB port.

\* Home Assistant's Role (Client): The pyseneye custom component running inside Home Assistant is now the sole party responsible for interpreting the data stream.

This separation creates immense resilience. If the Seneye device locks up and sends corrupted data, the USB/IP server on the Pi will dutifully forward that corrupted data without crashing. The pyseneye component in Home Assistant might fail to parse the data and report the sensor as "Unavailable," but this is a manageable, high-level application error. It will not crash the underlying connection. The USB/IP bridge will remain active, patiently waiting for the Seneye device to recover or be power-cycled. Once valid data is available again, the pyseneye component will resume its function, and the sensor readings will reappear.

By implementing this architecture, the problem is moved from a low-level, system-crashing instability to a high-level, gracefully-handled data availability issue. This shift is the key to achieving the long-term, hands-off stability required for a critical monitoring system.

Part 2: Forging the Server: Configuring the Raspberry Pi

This section provides the precise, step-by-step instructions for transforming the Raspberry Pi Zero 2W into a dedicated, resilient USB/IP server. Each command is presented as a distinct step to allow for verification along the way.

2.1 System Prerequisites

The process begins with a clean, updated foundation. It is confirmed that the Raspberry Pi has a fresh installation of Raspberry Pi OS Lite (64-bit) and has been fully updated using the command sudo apt update && sudo apt full-upgrade -y [User Query]. This ensures all base packages and kernel modules are current.

2.2 Installing the USB/IP Toolchain

The first step is to install the necessary user-space utilities that allow for the management of the USB/IP kernel modules.

\* Action: Install the usbip package.

\* Command:

sudo apt install usbip -y

\* Explanation: This command installs two key programs: usbip, the command-line tool used to list, bind, and attach devices, and usbipd, the daemon (background service) that listens for incoming client connections.

2.3 Identifying the Seneye Device

Before the device can be shared, the system needs to know how to identify it uniquely and reliably.

\* Action: Connect the Seneye Reef HUD to a USB port on the Raspberry Pi. Then, execute the following commands to find its hardware identifiers.

\* Command 1: Find the Vendor and Product ID.

lsusb

\* Explanation: This command lists all USB devices connected to the Pi. The output will contain a line for the Seneye device. It is crucial to locate this line and note the ID. For the Seneye Reef, this is typically 2437:0100. This ID is the device's unique hardware signature, burned into its firmware, and will never change.

\* Example Output:

Bus 001 Device 003: ID 2437:0100

\* Command 2: Find the Bus ID.

usbip list -l

\* Explanation: This command lists the same devices but in a format that usbip uses. It will show a busid for the Seneye device, such as 1-1.1. This busid represents the device's physical location on the USB bus. It is important to note that this ID can change if the device is plugged into a different USB port or if other devices are added or removed. This unreliability is a critical weakness that the next step will solve.

2.4 Building a Resilient systemd Service

Simply running the usbipd command manually is insufficient for a system that needs to run unattended for months. A systemd service is the modern Linux standard for managing daemons, ensuring they start on boot and recover from failure. This custom service file is the cornerstone of the server's stability.

\* Action: Create and edit a new systemd service file using the nano text editor.

\* Command:

sudo nano /etc/systemd/system/usbipd-seneye.service

\* Explanation: This command opens a blank text file. The entire block of text below should be copied and pasted into the nano window.

\* File Content (usbipd-seneye.service):

[Unit]

Description=USB/IP Host Daemon for Seneye Device

After=network.target

Wants=network.target

Type=forking

ExecStartPre=/bin/bash -c "until lsusb -d 2437:0100; do sleep 5; done"

ExecStart=/usr/sbin/usbipd -D

ExecStartPost=/bin/bash -c "/usr/sbin/usbip bind --busid=$(/usr/sbin/usbip list -p -l | grep '#usbid=2437:0100#' | cut -d'=' -f2 | cut -c -3)"

ExecStop=/bin/bash -c "/usr/sbin/usbip unbind --busid=$(/usr/sbin/usbip list -p -l | grep '#usbid=2437:0100#' | cut -d'=' -f2 | cut -c -3)"

Restart=always

RestartSec=5

[Install]

WantedBy=multi-user.target

\* Detailed Breakdown of the Service File's Power: This file contains several directives specifically designed to create a robust, self-healing service.

\* After=network.target: This dependency ensures that the service will not attempt to start until the Raspberry Pi has successfully connected to the network.

\* ExecStartPre=...: This is a critical resilience feature. It executes a small shell command before starting the main daemon. The command until lsusb -d 2437:0100; do sleep 5; done creates a loop that checks for the presence of the Seneye device every 5 seconds. The service will not proceed until the device is detected. This elegantly handles scenarios where the Pi boots faster than the Seneye device can initialize, preventing startup failures.

\* ExecStartPost=...: This is the most crucial command for reliability. Instead of using a static, unreliable busid, this command dynamically finds the correct one at runtime. It works by:

\* usbip list -p -l: Listing all local devices in a parsable format.

\* grep '#usbid=2437:0100#': Filtering that list to find the line containing the Seneye's unique and unchanging hardware ID.

\* cut -d'=' -f2 | cut -c -3: Extracting just the busid (e.g., 1-1) from that line.

\* usbip bind --busid=...: Using the dynamically discovered busid to bind the device. This makes the entire setup immune to port changes or the addition of other USB devices, directly solving the instability of static busids.

\* Restart=always & RestartSec=5: These directives instruct systemd to constantly monitor the usbipd daemon. If the process ever crashes for any reason, systemd will automatically attempt to restart it after a 5-second delay.

\* Action: Save the file and exit nano by pressing Ctrl+X, then Y, then Enter.

\* Action: Enable and start the newly created service.

\* Commands:

\* sudo systemctl daemon-reload (Tells systemd to read the new service file).

\* sudo systemctl enable usbipd-seneye.service (Configures the service to start automatically on boot).

\* sudo systemctl start usbipd-seneye.service (Starts the service immediately).

\* Verification:

\* Command 1: Check Service Status.

sudo systemctl status usbipd-seneye.service

\* Explanation: The output should show the service as active (running). This confirms that the daemon has started successfully and the Seneye device has been bound.

\* Command 2: Verify Exportable Device.

usbip list -r 127.0.0.1

\* Explanation: This command queries the local USB/IP server. The output should list the Seneye device as an exportable device, confirming that the server is fully operational and ready to accept a connection from the Home Assistant client.

Part 3: Receiving the Signal: Configuring the Home Assistant Client

With the Raspberry Pi server broadcasting the Seneye's USB connection, the next phase is to configure Home Assistant to receive and utilize this signal. This requires a different approach due to the nature of the Home Assistant Operating System.

3.1 The Challenge: Home Assistant OS is a Managed Appliance

Home Assistant OS (HAOS) is designed as a secure and stable appliance. A key aspect of this design is that the underlying operating system is largely immutable. Users cannot simply install packages with apt, modify system files, or create systemd services directly, as these changes will not persist across reboots or updates. This is a deliberate design choice to ensure system integrity.

The officially sanctioned method for adding low-level functionality, such as loading kernel modules or interacting with hardware, is through the Home Assistant Add-on system. Add-ons are containerized applications that are granted specific, controlled privileges to interact with the host system. Therefore, to set up the client side of the USB/IP bridge, a dedicated add-on is required.

3.2 Selecting and Installing the USB/IP Client Add-on

Several community-developed add-ons provide USB/IP client functionality. These add-ons package the necessary usbip client tools and contain the logic to load the required vhci-hcd kernel module on the Home Assistant host. While early versions existed, such as irakhlin/hassio-usbip-mounter, a more recent and actively maintained add-on, cryptedx/ha-usbip-client, offers improved features and clearer documentation. This guide will use the cryptedx add-on for its robustness and ongoing support.

\* Action: Add the custom add-on repository to Home Assistant.

\* In the Home Assistant UI, navigate to Settings > Add-ons.

\* Click the ADD-ON STORE button in the bottom right.

\* Click the three-dots menu in the top right corner and select Repositories.

\* In the dialog box, paste the following URL and click ADD:

https://github.com/cryptedx/ha-usbip-client

\* Close the repositories dialog.

\* Action: Install and configure the USB/IP Client add-on.

\* Refresh the Add-on Store page. The "USB/IP Client" add-on will now appear in the list.

\* Click on the add-on and then click INSTALL. Wait for the installation to complete.

\* After installation, navigate to the Configuration tab of the add-on.

\* Replace the default configuration with the following, updating the IP address and bus\_id as required.

\* File Content (Add-on Configuration):

log\_level: info

discovery\_server\_address: 192.168.X.X

devices:

- server\_address: 192.168.X.X

bus\_id: 1-1

\* Explanation of Configuration:

\* log\_level: Set to info for normal operation. Can be changed to debug for troubleshooting.

\* discovery\_server\_address and server\_address: Replace 192.168.X.X with the static IP address of the Raspberry Pi server. It is highly recommended to assign a static IP or DHCP reservation to the Pi to prevent connection issues.

\* bus\_id: This is the bus ID of the Seneye device as seen by the server. To find the correct value, SSH into the Raspberry Pi and run the command usbip list -r 127.0.0.1. The output will show the busid (e.g., 1-1). Enter that value here. Because the Pi is a dedicated server for this one device, this ID should remain stable.

\* Action: Start the add-on and verify the connection.

\* Navigate to the Info tab of the add-on.

\* Turn OFF the "Protection mode" toggle. This is required for the add-on to load kernel modules and interact with the host's hardware layer.

\* Turn ON the "Start on boot" toggle to ensure the connection is re-established automatically after a Home Assistant restart.

\* Click START.

\* Navigate to the Log tab. After a few moments, the log should show messages indicating that it has successfully loaded the vhci-hcd module and attached the remote device. A successful connection is the final confirmation that the virtual USB bridge is active.

Part 4: Reading the Data: The pyseneye Integration

With the USB device now virtually connected to Home Assistant, the final step is to install the software "driver" that can understand the Seneye-specific protocol and translate its raw data into meaningful sensor values.

4.1 The pyseneye Custom Component

The Home Assistant community has produced a solution for this exact purpose. A developer known as mcclown has created a Python library named pyseneye and a corresponding Home Assistant custom component. This component is designed to communicate directly with a Seneye device connected via a local USB port—which, thanks to the USB/IP bridge, is exactly what Home Assistant now perceives.

This integration is the final piece of the puzzle. It allows for direct, local polling of the Seneye device, completely bypassing the need for the official Seneye cloud services, the expensive Seneye Web Server (SWS), or a continuously running Windows PC.

4.2 Installation via HACS (Home Assistant Community Store)

The easiest way to install this custom component is through the Home Assistant Community Store (HACS). It is assumed that HACS is already installed.

\* Action: Add the pyseneye custom repository to HACS.

\* In the Home Assistant UI, navigate to HACS from the sidebar.

\* Go to the Integrations section.

\* Click the three-dots menu in the top right corner and select Custom repositories.

\* In the dialog box, paste the following URL into the "Repository" field:

https://github.com/mcclown/home-assistant-custom-components

\* Select Integration from the "Category" dropdown menu.

\* Click ADD.

\* Action: Install the integration from HACS.

\* The custom repository will now be visible in the HACS Integrations list.

\* Click on the "Seneye" integration that has appeared.

\* Click the DOWNLOAD button in the bottom right.

\* Confirm the download in the next dialog.

\* After the download is complete, Home Assistant will prompt for a restart. It is essential to restart Home Assistant at this point for the integration to be loaded.

4.3 Configuration in configuration.yaml

Unlike many modern integrations that use UI-based configuration flows, the pyseneye component is configured using YAML in the main configuration.yaml file.

\* Action: Add the Seneye sensor platform to the configuration file.

\* Method: Use the "File editor" or "Samba share" add-on to access and edit the configuration.yaml file located in the main /config directory of Home Assistant.

\* File Content (configuration.yaml addition): Add the following lines to the file. If a sensor: section already exists, simply add the - platform: seneye line under it.

sensor:

- platform: seneye

\* Explanation: The configuration for this component is remarkably simple, as noted in its documentation. It does not require any specific device paths or addresses. The component is designed to automatically scan the available USB ports (including the virtual one created by the USB/IP add-on) and detect the connected Seneye device.

\* Action: After saving the changes to configuration.yaml, restart Home Assistant one final time. This can be done via Settings > System > Restart.

Upon restart, the pyseneye component will load, detect the virtual Seneye device, and begin creating the associated sensor entities.

Part 5: Verification, Automation, and Integration

With all components configured and running, this final section focuses on verifying the system's operation, leveraging the new data for automation, and integrating it into the existing smart home dashboard.

5.1 Confirming a Successful Integration

The ultimate confirmation of success is the appearance of live data within Home Assistant.

\* Action: Navigate to the entity list to find the new sensors.

\* Method: In the Home Assistant UI, go to Settings > Devices & Services > Entities.

\* Verification: In the search bar at the top of the entity list, type seneye. The system should display several new entities, including:

\* sensor.seneye\_temperature

\* sensor.seneye\_ph

\* sensor.seneye\_nh3 (Free Ammonia)

\* Explanation: The presence of these entities with current state values (e.g., a temperature in °C, a pH value, and an NH3 reading) confirms that the entire chain is functioning correctly: the Seneye is communicating with the Pi, the Pi is broadcasting the USB data, the HA add-on is receiving it, and the custom component is successfully parsing it. The data updates approximately every 30 minutes, as recommended by the device manufacturer.

5.2 Building Your First Automation

The primary motivation for this project is live monitoring and alerting. Now that the Seneye data is available as standard Home Assistant entities, creating powerful automations is straightforward.

\* Action: Create a critical alert for high ammonia levels.

\* Example Automation: This automation will trigger if the free ammonia (NH3) level rises above a dangerous threshold (e.g., 0.02 ppm) and send a high-priority notification to a mobile device.

\* Navigate to Settings > Automations & Scenes.

\* Click CREATE AUTOMATION and select "Create new automation".

\* Trigger:

\* Trigger type: Numeric state

\* Entity: sensor.seneye\_nh3

\* Above: 0.02

\* Actions:

\* Action type: Call service

\* Service: notify.mobile\_app\_<your\_device\_name>

\* Message: CRITICAL REEF ALERT: Ammonia level has exceeded 0.02 ppm! Current reading is {{ states('sensor.seneye\_nh3') }} ppm.

\* Title: High Ammonia Alert

\* Explanation: This simple automation demonstrates the power of the new integration. It directly addresses the core requirement of using the Seneye for live ammonia monitoring and connects the reef tank's status to the broader smart home notification system.

5.3 Unifying with APEX

A key goal was to create a unified dashboard for viewing all tank parameters. With the Seneye data now natively integrated into Home Assistant, this is easily achieved.

\* Action: Add the new Seneye sensors to an existing Home Assistant dashboard.

\* Method:

\* Navigate to the desired dashboard.

\* Enter "Edit Dashboard" mode.

\* Add a new card, such as an "Entities" or "Gauge" card.

\* Select the sensor.seneye\_ph, sensor.seneye\_temperature, and sensor.seneye\_nh3 entities.

\* Explanation: These new entities can now be displayed alongside any existing entities from the Neptune APEX A3 controller. This achieves the goal of a single pane of glass, providing a comprehensive, at-a-glance overview of the reef tank's health, combining data from both the APEX and the Seneye systems.

Part 6: Long-Term Maintenance and Troubleshooting Guide

A truly robust system is not just one that works, but one that is easy to diagnose and maintain when issues inevitably arise. This section provides a comprehensive guide to understanding the system's behavior and resolving common problems.

6.1 The Philosophy of a Resilient System

The architecture constructed in this guide is inherently resilient. The systemd service on the Raspberry Pi is designed to be self-healing, automatically restarting the USB/IP daemon if it crashes and patiently waiting for the Seneye device to appear on boot. The Home Assistant add-on is similarly designed to persistently attempt reconnection if the server becomes unavailable. In many common failure scenarios, such as a power outage or a temporary network drop, the system is designed to recover entirely on its own without any manual intervention.

6.2 Common Scenarios and Solutions

\* Scenario 1: The Seneye Device Locks Up

\* Symptom: The Seneye sensor entities in Home Assistant become "Unavailable" or their values stop updating. The logs for the pyseneye component may show communication or timeout errors.

\* Cause: This is the known hardware-level lock-up of the Seneye device, where it stops responding correctly to USB commands.

\* Solution: The simplest and most effective solution is to power-cycle the Seneye device. Unplug the Seneye's USB cable from the Raspberry Pi, wait 30 seconds, and plug it back in. The systemd service on the Pi is designed for this exact event; its ExecStartPre command will detect the device's reappearance, and the ExecStartPost command will automatically re-bind it to the USB/IP server. The Home Assistant add-on will then re-establish its connection. No reboot of the Pi or Home Assistant is necessary.

\* Scenario 2: The Raspberry Pi Loses Power or Wi-Fi

\* Symptom: The Seneye entities in Home Assistant become "Unavailable." The logs for the "USB/IP Client" add-on will show repeated "could not connect to host" errors.

\* Solution: Restore power or network connectivity to the Raspberry Pi. The usbipd-seneye.service is enabled to start on boot (systemctl enable), so it will automatically launch and begin sharing the device as soon as the Pi is back online and connected to the network. The Home Assistant add-on will continue its reconnection attempts and will restore the link once the server is available again. The system will recover automatically.

\* Scenario 3: After a Monthly Slide Change

\* Symptom: A new slide needs to be activated.

\* Cause: The pyseneye custom component does not implement the proprietary protocol for activating new pH/NH3 slides.

\* Solution: This is the one maintenance task that requires temporarily breaking the bridge. Once a month, the Seneye device must be unplugged from the Raspberry Pi and connected to a Windows or Mac computer running the official Seneye Connect software to activate the new slide. Once the slide is activated, the device can be unplugged from the computer and reconnected to the Raspberry Pi. As in Scenario 1, the system will automatically detect its return and resume normal operation.

6.3 Diagnostic Toolkit

To troubleshoot issues, it is helpful to check the health of each component in the communication chain.

\* On the Raspberry Pi (via SSH):

\* sudo systemctl status usbipd-seneye.service: Checks if the core server daemon is running.

\* lsusb: Confirms the OS can see the Seneye device at a hardware level.

\* usbip list -l: Confirms the device is available to usbip.

\* From Any Machine on the Network (including the Home Assistant Terminal Add-on):

\* ping <pi\_ip\_address>: Confirms basic network connectivity to the Pi.

\* usbip list -r <pi\_ip\_address>: The ultimate test of the server. Confirms the Pi is on the network AND the usbipd service is running and has successfully bound the device.

\* In Home Assistant:

\* Add-on Logs: Navigate to Settings > Add-ons > USB/IP Client > Log. This is the first place to check for client-side connection issues.

\* Core Logs: Navigate to Settings > System > Logs. Look for any errors or warnings containing the words "seneye" or "pyseneye" to diagnose issues with the custom component itself.

6.4 Troubleshooting Quick Reference Table

This table provides an at-a-glance guide for rapidly diagnosing and resolving the most common problems.

Table 1: Troubleshooting Quick Reference

| Symptom | Probable Cause | Diagnostic Command(s) | Solution |

|---|---|---|---|

| HA Add-on log shows "could not connect to host" | Network issue, Pi server offline, or usbipd service not running. | From HA Terminal: ping <pi\_ip\_address><br>On Pi: systemctl status usbipd-seneye.service | Check Pi power/Wi-Fi. Reboot Pi. Restart the usbipd-seneye.service on the Pi. Check firewall rules. |

| usbipd service fails to start on Pi | Seneye device not connected, or Vendor/Product ID in service file is incorrect. | On Pi: lsusb<br>On Pi: journalctl -u usbipd-seneye.service | Ensure Seneye is securely plugged in. Verify the ID in the service file (2437:0100) matches lsusb output. |

| Seneye entities are "Unavailable" in Home Assistant | USB/IP connection dropped, or Seneye custom component has an issue. | Check HA USB/IP Add-on logs.<br>Check HA Core logs for errors related to "seneye" or "pyseneye". | Restart the USB/IP Client Add-on. If the issue persists, restart Home Assistant. |

| Seneye readings are stale or not updating | The Seneye device has locked up (a known issue). | Observe the device for any error lights. Check HA logs for communication timeout errors. | Physically unplug the Seneye device from the Pi for 30 seconds, then plug it back in. The system will recover automatically. |