

START

[Overview](#)[Quick start](#)

READING A SETUP

[What you get back](#)[Values or full detail](#)[Setting fields](#)[Value types](#)[Show, hide and allow](#)

CHANGING SETTINGS

[Send changes](#)[Answers and errors](#)[Stored setups](#)

CODE

[Ready-made parsers](#)[Recipes](#)[Pitfalls](#)

SVANLINK API · GUIDE

Read and change instrument settings as JSON

`getSetupJSON` returns an instrument's setup as a tree of the names and values you see in the setup editor. `setSetupJSON` takes back only the settings you change, checks them against the instrument's rules and writes them. This guide shows how to read that tree, what every field means and how to change settings safely.

Reference [getSetupJSON](#) [setSetupJSON](#)

Works on a connected instrument or a setup stored on SvanLINK

Code in Python, JavaScript and curl

Code examples are printed in Python. To print another language, choose it at the top of the page first.

1

Read`getSetupJSON` gives the tree and a fingerprint.

2

Look up

Find the setting, what it accepts and when it applies.

3

Check`setSetupJSON` with `check: true` writes nothing.

4

WriteThe same request without `check`.

Quick start

Read the setup of instrument `123456`, preview one change, then write it. Every request is a `POST` of JSON to your SvanLINK's API address. `token` is the session token from the [login](#) request.

Python

```
import requests

SVANLINK = "http://192.168.1.10:8000/" # your SvanLINK
TOKEN = "userToken" # from the login request
SERIAL = "123456" # the instrument's serial number

def call(request, **params):
    reply = requests.post(SVANLINK, json={"Request": request, "Params": params, "token": TOKEN},
                          timeout=120).json()
    if reply["Status"] != "ok":
        raise RuntimeError(reply["StatusMessage"])
    return reply["Response"]

setup = call("getSetupJSON", device=SERIAL)
print(setup["settings"]["Measurement"]["Profile 1"]["Filter"]) # A

change = {"Measurement": {"Profile 1": {"Filter": "C"}}}
preview = call("setSetupJSON", device=SERIAL, fingerprint=setup["fingerprint"],
               settings=change, check=True)
print(preview["changed"]) # [{"path": 'Measurement/Profile 1/Filter', 'code': 'BI0', 'before':
'A', 'after': 'C'}]

result = call("setSetupJSON", device=SERIAL, fingerprint=setup["fingerprint"], settings=change)
print("measurement restarted:", result["restarted"])
```

BEFORE YOU WRITE

Writing to an instrument that is measuring stops the measurement and starts it again. The answer says so with `"restarted": true`. Preview with `check: true` first.

What you get back

The answer names three things. Groups are JSON objects nested in the same order as the setup editor's menus. A setting is text, exactly as the editor shows it. The example below is shortened: the SV 303's full tree has 187 settings in six top groups.

getSetupJSON · answer from SV 303 #123456 (shortened)

```
{
  "Request": "getSetupJSON",
  "Status": "ok",
  "Response": {
    "deviceType": 303,
    "fingerprint": "0a5e1f293d50844f",
    "settings": {
      "Function": { "Measurement Function": "1/3 Octave" },
      "Measurement": {
        "General Settings": {
          "Start Delay": "0s",
          "Integration Period": "1 h",
          "Rolling Time(1)": "30m"
        },
        "Profile 1": { "Filter": "A", "Peak Filter": "C", "Detector": "Fast" },
        "Logging": {
          "Logger Setup": { "Logger": "On", "Logger Step": "1 s", "Logger Name": "L105" }
        }
      },
      "Display": {},
      "Instrument": {},
      "Auxiliary Setup": {},
      "Communication": {}
    }
  }
}
```

- deviceType** The instrument type as a number, here `303`. `listSetups` reports the same type as text, `"303"`.
- fingerprint** A short digest of every value an edit can change. Send it back with `setSetupJSON`. If the setup changed in between, SvanLINK refuses the write instead of overwriting the newer setup. Date and time settings are not part of it.
- settings** The tree. Use every name exactly as it comes back: names depend on the instrument type and firmware. When two siblings would share a name, the name gets a suffix, such as `Rolling Time(1)` or `Level (XX)` with the instrument's code.

PATHS IN THIS GUIDE

A path lists the names from the top group down to the setting. This guide writes it as

`Measurement/Profile 1/Filter`, and SvanLINK reports changes the same way. Some names contain a slash themselves, such as `1/3 Octave Source`, so in code keep a path as a list of names.

Values or full detail

By default each setting is only its value. Add `"options": true` to the read and each setting becomes an object that also says what it accepts and when the editor shows it. Hidden settings are included too, so the answer is much larger: 441 settings instead of 187 for this SV 303.

Use the values tree to read or change a few known settings. Use full detail when you build a form, validate input before sending it, or need to know why a setting is hidden.

Default

```
"Logger Setup": {
  "Logger": "On",
  "Logger Step": "1 s"
}
```

"options": true

```
"Logger Step": {
  "code": "DB",
  "type": "msec",
  "value": "1 s",
  "raw": "1000",
  "visible": true,
  "shownWhen": "Logger is On",
  "baseVisible": true,
  "visibility": [
    { "when": { "param": "DA", "op": "==", "value": "0" }, "show": false }
  ],
  "description": "Specifies how often results will be saved in the logger file.",
  "hint": "e.g. 500 ms, 2 s, 1 m",
  "ranges": [
    { "label": "100 ms", "from": 100, "to": 100 },
    { "label": "125 ms", "from": 125, "to": 125 },
    { "label": "200 ms", "from": 200, "to": 200 },
    { "label": "500 ms", "from": 500, "to": 500 },
    { "label": "1 s ... 1 m, every 1 s", "from": 1000, "to": 60000, "step": 1000 },
    { "label": "1 m ... 1 h, every 1 m", "from": 60000, "to": 3600000, "step": 60000 }
  ]
}
```

The same setting, `Measurement/Logging/Logger Setup/Logger Step`, in both shapes. `DA` is the code of `Logger`; its raw value `0` means Off.

Setting fields

With `"options": true` every setting has `code`, `type`, `value` and `visible`. The other fields appear only when they apply.

FIELD	WHAT IT HOLDS	USE IT TO
code	The instrument's own code for the setting, such as <code>BI0</code> .	Store a reference that survives renamed groups. Codes are the same across languages.
type	<code>bool</code> , <code>enum</code> , <code>flags</code> , <code>text</code> , <code>numeric</code> or a time type (<code>sec</code> , <code>msec</code> , <code>ptime</code> , <code>svantime</code> ...).	Pick the input control. See Value types .
value	The value as a person reads it.	Show it, edit it, send it back. It is the only field SvanLINK reads on a write.
raw	The instrument's stored form, such as <code>"2"</code> for filter A.	Evaluate conditions. Never send it as the value.
codes	For a time of day, the codes of its hour, minute and second, in place of <code>raw</code> .	Evaluate conditions on a time of day.
visible	Whether the setup editor shows it now.	Hide settings that do not apply.
baseVisible visibility	The starting visibility and the rules that change it.	Recompute <code>visible</code> after a local edit. See Show, hide and allow .
shownWhen	The same rules in words, such as <code>Logger is On</code> .	Explain to a user why a setting is missing.
choices	<code>[{"label", "raw"}]</code> . A choice with <code>allowedIf</code> is legal only while that condition holds; <code>allowedWhen</code> says it in words.	Fill a dropdown or a set of checkboxes.
ranges	<code>[{"label", "from", "to", "step"}]</code> in the raw unit; <code>step</code> defaults to 1.	Validate a number or time before sending it.
text	A text's limits: <code>maxLength</code> or <code>maxBytes</code> , and the allowed <code>characters</code> .	Limit a text field.
hint	A unit (<code>Hz</code> , or <code>0.01dB</code> when the number counts in that unit) or a typing example.	Label or placeholder of the input.
accepts	What it takes when nothing above says, such as <code>any number (0.01dB)</code> .	Show as help text.
description	The help text from the instrument's documentation.	Show as help text.
readOnly	<code>true</code> when the value cannot be set.	Show the value without an input.

Value types

Always read and send the readable form in `value`. SvanLINK converts it to the instrument's form and refuses anything the instrument would not accept. Every row below is a real SV 303 setting, and every value in the last column was accepted.

TYPE	SETTING	AS READ	HOW TO WRITE IT	EXAMPLE
bool	Speaker check schedule/System Check	Off	One of the two choice labels.	On
enum	Profile 1/Filter	A	One choice label: Z, A, C, B, LF.	C
flags	Logger Results/Profile 1	Lpeak, Lmax, Lmin, Leq, LR(1), LR(2)	Labels joined with a comma and a space. raw is the sum of the chosen bits: 111 = 1+2+4+8+32+64.	Lpeak, Lmax, Leq
text	Logger Setup/Logger Name	L105	At most 8 characters from 0-9, A-Z, _ and space.	NIGHT_1
numeric	Calibration/Level	11400	A whole number in the unit of hint , here 0.01 dB.	11400 = 114.00 dB
sec	General Settings/Start Delay	0s	A duration such as 30 s, 5m or 1h, inside ranges (seconds).	10s
msec	Logger Setup/Logger Step	1 s	A duration inside ranges (milliseconds). 1.5 s is refused.	2 s
svantime	General Settings/Integration Period	1 h	A duration or infinity .	15m
ptime	Logger Setup/Splitting Time 1 (hh:mm)	00:00	A time of day, HH:MM. It has codes instead of raw .	06:30
time_shift	Speaker check schedule/Time	00:00	HH:MM between 00:00 and 23:59.	05:30

Other instruments add types such as **min** and **time_min**. They follow the same pattern: read **hint** and **ranges**, send readable text.

Show, hide and allow

Some settings only matter while another one has a certain value. **Logger Step** matters only while **Logger** is On. With **"options": true** each setting carries these rules, so you can show the same form the setup editor shows and update it while the user edits.

To work out whether a setting is shown:

1. Collect the raw value of every setting by its code: **raw**, or for a time of day each part of **value** under its **codes**.
2. Start from **baseVisible**.
3. Go through **visibility** in order. When a rule's **when** holds, set the result to its **show**. The last rule that holds wins.

A condition is **{"param", "op", "value"}**, or **{"and": [...]}** or **{"or": [...]}** around more conditions, or plain **true** or **false** when SvanLINK has already worked it out. **param** is a code. **==** and **!=** compare the raw values as text; **<**, **>**, **<=** and **>=** compare them as whole numbers.

Choices use the same conditions. On this SV 303, **Measurement Function** offers **1/1 Octave** only while the instrument's hidden **1/3 Octave Locked** setting (**YE**) is 1. It is 0, so SvanLINK refuses that choice and says why.

Python

```
def holds(condition, raw):
    if isinstance(condition, bool):
        return condition
    if "and" in condition:
        return all(holds(c, raw) for c in condition["and"])
    if "or" in condition:
        return any(holds(c, raw) for c in condition["or"])
    current, op, value = raw.get(condition["param"], condition["op"], condition["value"])
    if current is None:
        return False
    if op in ("==", "!="):
        return (str(current).strip() == value) == (op == "==")
    try:
        left, right = int(current), int(value)
    except ValueError:
        return False
    return {"<": left < right, ">": left > right, "<=": left <= right, ">=": left >= right}[op]

def is_visible(setting, raw):
    shown = setting.get("baseVisible", setting["visible"])
    for rule in setting.get("visibility", []):
        if holds(rule["when"], raw):
            shown = rule["show"]
    return shown
```

Send changes

`setSetupJSON` takes the setting you change in one of three forms. In each case SvanLINK compares with the setup it reads from the instrument, writes only what differs and leaves every other setting as it was.

A branch of the tree

Send the groups down to the settings you change, with the new values as text. Several settings, even in different groups, go in one request. Related settings are checked together against the setup as it will be after all of them, so switching the logger on and setting its step in one request works.

Params

```
{
  "device": "123456",
  "fingerprint": "0a5e1f293d50844f",
  "settings": {
    "Measurement": {
      "Profile 1": { "Filter": "C" },
      "Logging": { "Logger Setup": { "Logger": "On", "Logger Step": "500 ms", "Logger Name":
"NIGHT_1" } }
    }
  }
}
```

Objects from a full-detail read

A setting may also be `{"value": "C"}`. That is the shape `"options": true` returns, so a form can send its edited objects back unchanged. Only `value` is read.

Params

```
{
  "device": "123456",
  "fingerprint": "0a5e1f293d50844f",
  "settings": { "Measurement": { "Profile 1": { "Filter": { "code": "BI0", "type": "enum", "value":
"C" } } } }
}
```

One setting by name

Send `name` and `value` instead of `settings`. `name` can be the setting's name when it is unique, its code, or its group and name.

Params · three ways to name the same kind of change

```
{ "device": "123456", "name": "BI0", "value": "C" }
{ "device": "123456", "name": "Profile 1/Filter", "value": "C" }
{ "device": "123456", "name": "Logger Step", "value": "2 s" }
```

`Filter` alone is refused on this instrument: four settings have that name, and the error lists them with their codes.

Check, then write

- Always send the `fingerprint` of the read you based the change on.
- Send the request with `"check": true`. It answers with `changed` and writes nothing.
- Send the same request without `check` to write it.

ALL OR NOTHING

If any name is unknown or any value is refused, nothing is written, not even the settings that were fine. Fix the request and send all of it again.

Answers and errors

After a write

setSetupJSON · answer

```
{
  "Request": "setSetupJSON",
  "Status": "ok",
  "Response": {
    "changed": [
      { "path": "Measurement/Profile 1/Filter", "code": "BI0", "before": "A", "after": "C" },
      { "path": "Measurement/Logging/Logger Setup/Logger Step", "code": "DB", "before": "1 s",
"after": "2 s" }
    ],
    "restarted": true
  }
}
```

changed One entry per setting that now differs, with its path, code and readable values before and after.

restarted `true` when a running measurement was stopped for the write and started again. `false` for a check, for a stored setup and when nothing needed writing.

message Only when nothing was written: `Check only: nothing was written`, or `Every value already matches the device`.

Errors

On an error `Status` is `"error"` and `StatusMessage` says why. Several problems are joined into one message. These are real messages from the SV 303.

STATUSMESSAGE	WHAT TO DO
The setup on the device changed since it was read. Read it again with <code>getSetupJSON</code> and redo the changes.	Someone changed the setup after your read. Read again, apply your change to the new tree, send the new fingerprint.
Unknown setting: Measurement/Profile 1/Filter	The name is not in this instrument's setup. Copy names from the tree you read.
Measurement/Profile 1/Filter: 'Filter' has no option 'X'. Allowed: Z; A; C; B; LF	Send one of the listed labels.
Measurement/Logging/Logger Setup/Logger Step: 'Logger Step': expected a time like 500 ms, 2 s or 1 m – got '1.5 s'	Send a value inside <code>ranges</code> .
Measurement/Logging/Logger Setup/Logger Name: at most 8 characters, got 11	Keep within <code>text.maxLength</code> and <code>text.characters</code> .
Function/Measurement Function: can't be '1/1 Octave' unless 1/3 Octave Locked is 1	That choice is not allowed with the instrument's current settings. Offer only the choices whose <code>allowedIf</code> holds.
Device 123456 is not connected	Check the serial number and that the instrument is connected to SvanLINK.
Failed to download setup from device (timeout or connection error)	The instrument did not answer in time. Try again.
Either "device", or "setup" and "type", are required	Send exactly one target: a connected instrument, or a stored setup.

Stored setups

Both requests also work on a setup file stored on SvanLINK. Send `setup` (its name) and `type` (as `listSetups` reports it) instead of `device`. `setSetupJSON` then saves the edited file over the stored one and no instrument is touched.

Params

```
{
  "setup": "Night survey",
  "type": "303",
  "settings": { "Measurement": { "Logging": { "Logger Setup": { "Logger Step": "500 ms" } } } }
}
```

Ready-made parsers

Two small modules that do the work of this guide. Copy one into your project. Both have no dependencies apart from `requests` in Python.

- `read()` returns a `Setup` that walks the tree, looks up settings by path, name or code, and works in either shape.
- `raw_values()`, `is_visible()` and `allowed_choices()` apply the rules from [Show, hide and allow](#).
- `patch()` turns `{path: value}` into the nested tree. `change()` sends it with the fingerprint of the read.

svanlink_setup.py

```
"""Read, inspect and change instrument settings through SvanLINK's setup JSON.

from svanlink_setup import SvanLink
```

```

svanlink = SvanLink("http://192.168.1.10:8000", token="userToken")
setup = svanlink.read("123456")          # getSetupJSON
print(setup.get("Measurement/Profile 1/Filter"))    # A

change = {"Measurement/Profile 1/Filter": "C"}
print(svanlink.change(setup, change, check=True))  # what would change
svanlink.change(setup, change)                    # write it

```

Needs Python 3.8+ and `requests`.

```

"""
import requests

class SvanLinkError(Exception):
    """SvanLINK answered with Status "error". The message says why."""

def _path(path):
    """'Measurement/Profile 1/Filter' or a tuple of names -> a tuple of names."""
    return tuple(path.split("/")) if isinstance(path, str) else tuple(path)

def _is_setting(node):
    # In the values tree a setting is text. With "options": true it is an
    # object that has a code and a type. A group is any other object.
    return not isinstance(node, dict) or ("code" in node and "type" in node)

def patch(changes):
    """{"Measurement/Profile 1/Filter": "C", ...} -> the nested tree setSetupJSON takes.

    A key can also be a tuple of names, which you need for names that contain
    "/" (such as "1/3 Octave Source"); find() returns paths in that form."""
    tree = {}
    for path, value in changes.items():
        *groups, name = _path(path)
        node = tree
        for group in groups:
            node = node.setdefault(group, {})
        node[name] = value
    return tree

def holds(condition, raw):
    """Whether a visibility `when` or a choice's `allowedIf` holds for these raw values."""
    if isinstance(condition, bool):          # already worked out by SvanLINK
        return condition
    if "and" in condition:
        return all(holds(c, raw) for c in condition["and"])
    if "or" in condition:
        return any(holds(c, raw) for c in condition["or"])
    current = raw.get(condition["param"])
    if current is None:
        return False
    op, value = condition["op"], condition["value"]
    if op == "==":
        return str(current).strip() == value
    if op == "!=":
        return str(current).strip() != value
    try:
        left, right = int(current), int(value)
    except (TypeError, ValueError):
        return False
    return {"<": left < right, ">": left > right, "<=": left <= right, ">=": left >= right}[op]

class Setup:
    """One getSetupJSON answer."""

    def __init__(self, response, serial=None):

```

```

        self.serial = serial
        self.device_type = response["deviceType"]
        self.fingerprint = response["fingerprint"]
        self.tree = response["settings"]

    def items(self):
        """(path, setting) for every setting. A path is a tuple of names."""
        def walk(node, path):
            for name, child in node.items():
                if _is_setting(child):
                    yield path + (name,), child
                else:
                    yield from walk(child, path + (name,))
        return walk(self.tree, ())

    @property
    def detailed(self):
        """True when read with "options": true."""
        return any(isinstance(setting, dict) for _, setting in self.items())

    def get(self, path):
        """The setting at a path: text, or an object when detailed."""
        node = self.tree
        for name in _path(path):
            node = node[name]
        return node

    def value(self, path):
        """The readable value at a path, in either form."""
        setting = self.get(path)
        return setting["value"] if isinstance(setting, dict) else setting

    def find(self, name):
        """Paths of every setting with this name, or with this code when detailed."""
        return [path for path, setting in self.items()
                if path[-1] == name or (isinstance(setting, dict) and setting.get("code") == name)]

    def raw_values(self):
        """{code: raw} for every setting, the values conditions compare. Detailed only."""
        raw = {}
        for _, setting in self.items():
            if "raw" in setting:
                raw[setting["code"]] = setting["raw"]
            elif "codes" in setting:
                # a time of day: hour, minute, second
                for code, part in zip(setting["codes"], setting["value"].split(":")):
                    raw[code] = str(int(part))
        return raw

    def is_visible(self, setting, raw=None):
        """Whether the editor shows a setting, for these raw values (default: as read)."""
        if "visibility" not in setting:
            return setting["visible"]
        raw = self.raw_values() if raw is None else raw
        shown = setting["baseVisible"]
        for rule in setting["visibility"]:
            # in order; the last rule that holds wins
            if holds(rule["when"], raw):
                shown = rule["show"]
        return shown

    def allowed_choices(self, setting, raw=None):
        """The choices that are legal now (a choice with allowedIf only while it holds)."""
        raw = self.raw_values() if raw is None else raw
        return [c for c in setting.get("choices", []) if holds(c.get("allowedIf", True), raw)]

class SvanLink:
    """A SvanLINK host. `url` is its API address, e.g. http://192.168.1.10:8000"""

    def __init__(self, url, token=None, timeout=120):
        self.url = url.rstrip("/") + "/"

```

```

self.token = token
self.timeout = timeout    # reading and writing a setup takes seconds

def call(self, request, **params):
    body = {"Request": request, "Params": params}
    if self.token:
        body["token"] = self.token
    reply = requests.post(self.url, json=body, timeout=self.timeout)
    reply.raise_for_status()
    data = reply.json()
    if data.get("Status") != "ok":
        raise SvanLinkError(data.get("StatusMessage") or request + " failed")
    return data["Response"]

def read(self, serial, options=False):
    """The connected instrument's current setup."""
    return Setup(self.call("getSetupJSON", device=serial, options=options), serial)

def change(self, setup, changes, check=False):
    """Write {path: value} to the instrument `setup` was read from.

    Refused if its setup changed since that read. check=True only reports
    what would change. A running measurement is stopped for the write and
    started again ("restarted": true)."""
    return self.call("setSetupJSON", device=setup.serial, fingerprint=setup.fingerprint,
                    settings=patch(changes), check=check)

```

CHECKED

Both modules were run against an SV 303 setup: 187 settings in the values tree and 441 in full detail, visibility the same as SvanLINK's for every setting, and patches that SvanLINK's own code accepts.

Recipes

Each recipe uses the ready-made parser, with `svanlink` created as in its first lines.

Use the same filter on every profile

Python

```

setup = svanlink.read("123456")
changes = {path: "A" for path in setup.find("Filter") if path[-2].startswith("Profile")}
print(svanlink.change(setup, changes, check=True)["changed"])
svanlink.change(setup, changes)

```

Apply one change to several instruments

Read each instrument first. Each read has its own fingerprint, and names can differ between instrument types, so look the setting up in every tree.

Python

```

for serial in ["123456", "168718", "45123"]:
    setup = svanlink.read(serial)
    paths = setup.find("Logger Step")
    if len(paths) != 1:
        print(serial, "skipped: Logger Step found", len(paths), "times")
        continue
    result = svanlink.change(setup, {paths[0]: "1 s"})
    print(serial, result["changed"] or "already 1 s", "(restarted)" if result["restarted"] else "")

```

List what each setting accepts

A starting point for a form: every setting the editor shows now, with the values it takes.

Python

```

setup = svanlink.read("123456", options=True)
raw = setup.raw_values()
for path, s in setup.items():
    if not setup.is_visible(s, raw) or s.get("readOnly"):
        continue
    accepts = ([c["label"] for c in setup.allowed_choices(s, raw)]
               + [r["label"] for r in s.get("ranges", [])])
    print(" > ".join(path), "=", s["value"], "| accepts:",
          "; ".join(accepts) or s.get("accepts") or s.get("hint", ""))

```

Output, first lines

```

Function > Calibration > Speaker check schedule > System Check = Off | accepts: Off; On
Function > Calibration > Level = 11400 | accepts: any number (0.01dB)
Function > Calibration > Max. file size = 10 | accepts: No limit; 1 ... 255
Function > Calibration > Post Calibration = Files after last calibration | accepts: Off; Last file;
Files after last calibration
Function > Calibration > Auto Calibration = Off | accepts: Off; On
Function > Measurement Function = 1/3 Octave | accepts: Level Meter; 1/3 Octave
Measurement > General Settings > Start Delay = 0s | accepts: 0s ... 1m; 1m ... 1h, every 1m
Measurement > General Settings > Start Sync. = 1s | accepts: Off; 1s; 1m; 15m; 30m; 1h
Measurement > General Settings > Integration Period = 1 h | accepts: infinity ... inf.
... 178 more

```

Pitfalls

Names belong to one instrument type

Group and setting names come from the instrument type and its firmware. Read before you write and look settings up in that tree. If you must store a reference, store the `code`.

A name can contain a slash

`Measurement/Measurement Trigger/1/3 Octave Source` is three groups and one name, not five names. Keep paths as lists of names in code.

A name can repeat

The SV 303 has `Filter` in Profile 1, 2 and 3 and in Spectrum. In a tree that is never a problem. With `name` and `value`, use the code or the group and name.

Hidden settings

The values tree leaves out settings the editor hides. Full detail includes them with `visible: false`. You can still set them.

Send value, compare raw

Send the readable `value`. `raw` is for evaluating conditions only.

Give the requests time

A read downloads the setup from the instrument and a write uploads it, each over the instrument's own link. Allow a minute before timing out.

SvanLINK · Setup JSON guide. The examples use the setup of an SV 303 (#123456). Full details of every parameter are in the [API documentation](#).