



OPEN AQUARIUM PROJECT SPECIFICATION

Automation

PRE-DRAFT

Revision	0.3
Status	Pre-Draft. A candidate document submitted for consideration.
Date	2026-10-18
Identifier	automation
Authors	Trent Scholl, Project Lead, Open Aquarium Project
Editors	Trent Scholl
License	Community Specification License 1.0

Pre-Draft. This is a proposed initial draft. The Working Group has not approved it, and it will change. Comment by opening an issue with the `automation` label.

1 License

This Specification is subject to the Community Specification License 1.0 (the “Specification License”), available at <https://github.com/CommunitySpecification/1.0> and in `LICENSES/Community-Spec-1.0.md`. Contributions to this Specification are made under the Community Specification Contributor License Agreement 1.0.

The JSON Schema, presets, and example files published with this Specification are licensed under the Apache License 2.0 (`LICENSES/Apache-2.0.txt`).

The Scope that bounds the patent commitments made under the Specification License is set out in `Scope.md`. Exclusion notices, license acceptances, and withdrawals are recorded in `Notices.md`.

Notes:

1. Appendices are informative (non-normative) unless an appendix is explicitly marked normative.

THESE MATERIALS ARE PROVIDED “AS IS.” The Contributors and Licensees expressly disclaim any warranties (express, implied, or otherwise), including implied warranties of merchantability, non-infringement, fitness for a particular purpose, or title, related to the materials. The entire risk as to implementing or otherwise using the materials is assumed by the implementer and user. IN NO EVENT WILL THE CONTRIBUTORS OR LICENSEES BE LIABLE TO ANY OTHER PARTY FOR LOST PROFITS OR ANY FORM OF INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY CHARACTER FROM ANY CAUSES OF ACTION OF ANY KIND WITH RESPECT TO THIS DELIVERABLE OR ITS GOVERNING AGREEMENT, WHETHER BASED ON BREACH OF CONTRACT, TORT (INCLUDING NEGLIGENCE), OR OTHERWISE, AND WHETHER OR NOT THE OTHER MEMBER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

2 Compliance with Open Aquarium Tenets

2.1 Openness

The Specification is published under the Community Specification License 1.0, and its schema, presets, and examples under the Apache License 2.0. The configuration format is plain JSON with a published schema and validator, so anyone can write a controller, an editor, or a converter for it without permission or payment. The format is declarative: it describes what equipment should do, and contains no program code, so reading it needs no particular product.

2.2 Interoperability

An owner’s configuration, including functions, interlocks, modes, step programs, and their current state, moves between controllers from different makers. Configurations refer to equipment by Data Model equipment and Device Classes endpoint, not by product, so the same configuration runs whatever hardware carries it out. Execution is defined precisely enough that two conforming controllers given the same configuration and the same inputs make the same decisions (Section 17.6). Values use Data Model parameters and units, outputs use Device Classes, and failures use the terms of the Fail-Safe Behavior specification.

2.3 Local Control

Every element of a configuration runs on the controller, with no network service, account, or vendor. Schedules use the controller’s own clock, and sun and moon times are computed on the controller. Owners can export their complete configuration and state at any time, and import them into another controller (Section 19).

2.4 Safety

Every control function defines what it does when an input is unavailable, and every input is checked for age (Sections 9.8 and 11.9). Interlocks, and `stop-all`, a person's emergency stop, can only stop equipment, and they take precedence over everything else, including the owner's overrides and equipment that a person runs at its own controls (Sections 17.3 and 17.4). A heater and a chiller serving one system are kept from running together (Section 13.5). A controller warns its owners, until it is fixed, while a safety function, such as a heater's cutoff or the response to a leak, depends on a radio link alone (Section 13.7). Every mode ends: at the end of its duration, when a condition ends it, or as its kind defines (Section 14.1). Temporary output states, such as a pump paused for feeding, an override, or a raised temperature during a treatment, are sent with `revertAfter`, so they end even if the controller fails (Sections 11.7 and 17.5). Stops with no known end, such as an interlock's stop of a return pump while a leak sensor is wet, are latched, so they last even if the controller fails (Section 17.5). Modes and `stop-all` leave alarm outputs to their alarm function, so that a siren keeps signaling critical alarms (Sections 14.4 and 17.3). Setpoints that change over time stay within limits the owner sets (Section 12.4). After a restart, programs resume from the clock, temporary states last no longer than the time they had left, and `stop-all` lasts until it expires (Section 17.7).

2.5 Sustainability

A configuration outlives the controller it was written on, so owners can replace or repair a controller without rebuilding years of tuning. Presets for common equipment protect it from short cycling (Section 18), and power-loss mode makes batteries last longer by stopping what is not needed. Because programs are tied to the equipment's role rather than to a module, a replaced module picks up its programming.

Revision History

Revision	Date	Description	Authors
0.1	2026-10-14	Initial Pre-Draft.	Trent Scholl
0.2	2026-10-16	Every mode ends: modes of extension kinds need a <code>duration</code> or an <code>end.when</code> ; a standing warning while a safety function has no path that uses no radio link; <code>stop-all</code> at the precedence of interlocks, and in the exception for local control; power-loss modes leave an output whose battery state is <code>function</code> to its function; default maximum ages that follow measurement intervals and the binding's reporting interval, with a warning for shorter ones; lost devices' outputs, whose state is unknown, in interlocks; exclusive interlocks between groups of outputs; resuming control with no demand; temporary states and reverts as Device Classes revision 0.3 defines them; <code>emergency-mode</code> and <code>safety-path-missing</code> among the faults; override <code>by</code> as a user's UUID; a start-up that is never an edge; a reference method for moon events; the operations that each action needs; a test procedure, mapped to its requirements, for every requirement; references cited by tag; one sentence per line.	Trent Scholl
0.3	2026-10-18	The Project Lead's decisions of 17 October 2026 applied: stops with no known end that interlocks and emergency modes send are latched, and a <code>requires</code> interlock's stop while the output it requires is paused carries that pause's end; temporary setpoints from offests and step programs sent with <code>set</code> and <code>revertAfter</code> ; the function types <code>alarm</code> , <code>top-off</code> , <code>water-change</code> , <code>ro-di</code> , <code>dosing</code> , <code>water-testing</code> , <code>lighting</code> , and <code>flow</code> , which other specifications define, with their saved state as <code>state.functions["<function id>"]</code> , in place of the provisional names, which controllers still read; the checks of top-off, water change, and RO/DI functions at the precedence of interlocks, and water-change modes of different systems active together; sun and moon events by the method <code>oap-1</code> of the Lighting Programs specification, with <code>noonAt</code> , <code>shiftDays</code> , latitudes up to 60°, and sun events within 2 s of it; who started each mode recorded; <code>stop-all</code> kept across restarts; modes and <code>stop-all</code> leave alarm outputs to their alarm function; the stops of power-loss modes, of other specifications' checks, and of a heater's or chiller's own cutoff sent to equipment under local control; safety functions judged as Fail-Safe Behavior revision 0.4 judges them, with a system's leak response as one; the precedence of Table 25, presets without targets, and one conformance level, no longer provisional; the fan preset with the equipment category <code>fan</code> ; the systems and equipment of documents of revision 0.1 read as Data Model revision 0.4 says. Also: a lost output with <code>continue</code> counts its pending revert; only stops sent during a power-on delay, and no alarm for what it refuses; analyzer inputs that read accepted, corrected results, and observation	Trent Scholl

Revision	Date	Description	Authors
		inputs that pass over what the Data Model says not to act on; every hold of the Probe Interface specification named; mode commands that only a person starts forbidden; overrides of revision 0.1 read without a by that is not a UUID; the schema's \$id https://openaquariumproject.org/schemas/automation/0.3 ; "A pass criterion that checks what a SHOULD asks applies to a product that does it"; corrected test procedures, and a test for each new requirement; references to the final revisions of the other specifications; issues 2, 7, 12, 14, 15, 19, and 29 deferred beyond revision 1.0.	

Contents

1	License	2
2	Compliance with Open Aquarium Tenets	2
2.1	Openness	2
2.2	Interoperability	2
2.3	Local Control	2
2.4	Safety	3
2.5	Sustainability	3
	Revision History	3
3	Scope	8
4	References	8
4.1	Normative References	8
4.2	Informative References	9
5	Terms, Definitions, and Conventions	9
5.1	Requirements Language	9
5.2	Terms and Definitions	9
5.3	Notation	10
6	Introduction	10
6.1	Purpose	10
6.2	Design Principles	11
6.3	Model Overview	11
6.4	Example	12
7	Configuration Documents	13
7.1	Encoding	13
7.2	Top-Level Members	13
7.3	Identifiers and References	14
7.4	Systems and Equipment	14
7.5	Numbers and Units	14
7.6	Times, Dates, and Durations	14
7.7	Extensions	14
7.8	Revisions and Compatibility	15
8	Outputs	15
8.1	Output Members	15
8.2	Output States	15
8.3	Output Classes	16
8.4	Virtual Outputs	16
8.5	Bindings	17
9	Inputs	17
9.1	Input Types	17
9.2	Endpoint Inputs	17
9.3	Output Inputs	18
9.4	Fault Inputs	19
9.5	Observation Inputs	19
9.6	Moon Illumination	19
9.7	Derived Inputs	19
9.8	Availability and Maximum Age	20
10	Conditions	20
10.1	Values and Unknown Values	20
10.2	Comparisons	21
10.3	All, Any, and Not	21
10.4	Latches	21
10.5	Timers	22
10.6	Schedules	22
10.7	Sun and Moon	22
10.8	Mode and Unavailable Conditions	23
11	Control Functions	23
11.1	Common Members	23
11.2	Fixed	24

11.3	Condition	24
11.4	Setpoint Control	24
11.5	Heating and Cooling Pairs	25
11.6	Proportional Control	25
11.7	Device Setpoints	25
11.8	Timing	26
11.9	Unavailable Inputs	26
11.10	Functions Defined by Other Specifications	27
12	Setpoints	28
12.1	Setpoint Sources	28
12.2	Daily, Seasonal, and Dated Setpoints	28
12.3	Offsets	28
12.4	Limits	28
12.5	When the Time Is Unavailable	29
13	Interlocks	29
13.1	General	29
13.2	Block	29
13.3	Requires	29
13.4	Exclusive	30
13.5	Heaters and Chillers	30
13.6	Leaks and Low Water	30
13.7	Safety Functions	30
14	Modes	31
14.1	Mode Kinds	31
14.2	Mode Members	32
14.3	Starting and Ending Modes	32
14.4	Output Actions	32
14.5	Mode Commands	32
14.6	Power-Loss Mode	33
14.7	Emergency Mode	33
15	Overrides	33
16	Step Programs	34
16.1	Program Members	34
16.2	Steps and Ramps	34
16.3	Starting, Running, and Stopping	35
16.4	Resuming After an Outage	35
16.5	Programs Together	35
17	Execution	35
17.1	Evaluation Cycle	35
17.2	Evaluation Order	35
17.3	Precedence	35
17.4	Commands	36
17.5	Temporary States, revertAfter, and Latched Stops	37
17.6	Determinism	38
17.7	Start-Up and Restart	38
17.8	Saved State	39
17.9	Faults	39
17.10	Visibility	39
18	Presets	40
18.1	Using Presets	40
18.2	Preset Tables	40
19	Portability	44
19.1	Export	44
19.2	Import	44
19.3	Sharing	44
20	Conformance	44
20.1	Configuration Documents	44
20.2	Controllers	45
20.3	Schema and Tools	45
20.4	Test Procedures	45

21 Security Considerations 50

Appendix A — Approval Checklist 51

Appendix B — Examples (Informative) 52

Appendix C — Commercial Programming Models (Informative) 52

Appendix D — Open Issues (Informative) 52

3 Scope

This Specification defines:

- a model of automation: control functions that set outputs from inputs, kept separate from the equipment that carries them out;
- inputs, conditions, and logic, including schedules and sun and moon times;
- control functions: fixed states, conditions, setpoint control with hysteresis, proportional control, and setpoints sent to self-regulating equipment;
- setpoints that change over time, timing limits, and behavior when inputs are unavailable;
- interlocks between outputs, global modes, manual overrides, and step programs;
- how a controller evaluates a configuration, and which element takes precedence;
- presets for common equipment; and
- a portable JSON format for a complete configuration and its state.

This Specification does not define:

- alarm, top-off, water-change, RO/DI, dosing, water-testing, lighting, or flow functions, which their own specifications define as function types that this Specification hosts (Section 11.10);
- how faults become alarms and notifications, which the Alarms and Notifications specification defines [OAP-AN];
- fail-safe behavior of devices, which the Fail-Safe Behavior specification defines [OAP-FS];
- how configurations are transferred, which the Controller API specification defines [OAP-CA], or who may change them, which the Security and Access specification defines [OAP-SA]; or
- recommended values for water parameters.

4 References

4.1 Normative References

- **[RFC 2119]** Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- **[RFC 8174]** Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, May 2017.
- **[RFC 8259]** Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, December 2017.
- **[RFC 3339]** Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, July 2002.
- **[JSON Schema]** "JSON Schema: A Media Type for Describing JSON Documents" and "JSON Schema Validation", draft 2020-12, <https://json-schema.org/draft/2020-12>.
- **[TZDB]** Internet Assigned Numbers Authority, "Time Zone Database", <https://www.iana.org/time-zones>.
- **[OAP-DM]** Open Aquarium Project, "Open Aquarium Data Model", Revision 0.4, [specs/data-model/spec.md](https://open-aquarium.org/specs/data-model/spec.md).
- **[OAP-DC]** Open Aquarium Project, "Open Aquarium Device Classes", Revision 0.5, [specs/device-classes/spec.md](https://open-aquarium.org/specs/device-classes/spec.md).
- **[OAP-FS]** Open Aquarium Project, "Open Aquarium Fail-Safe Behavior", Revision 0.4, [specs/fail-safe-behavior/spec.md](https://open-aquarium.org/specs/fail-safe-behavior/spec.md).
- **[OAP-MB]** Open Aquarium Project, "Open Aquarium Module Bus", Revision 0.4, [specs/module-bus/spec.md](https://open-aquarium.org/specs/module-bus/spec.md).
- **[OAP-DP]** Open Aquarium Project, "Open Aquarium Device Protocol", Revision 0.3, [specs/device-protocol/spec.md](https://open-aquarium.org/specs/device-protocol/spec.md).
- **[OAP-SA]** Open Aquarium Project, "Open Aquarium Security and Access", Revision 0.3, [specs/security-and-access/spec.md](https://open-aquarium.org/specs/security-and-access/spec.md).
- **[OAP-LP]** Open Aquarium Project, "Open Aquarium Lighting Programs", Revision 0.3, [specs/lighting-programs/spec.md](https://open-aquarium.org/specs/lighting-programs/spec.md).
- **[POSIX]** IEEE Std 1003.1-2024, "The Open Group Base Specifications Issue 8", the TZ environment variable, <https://pubs.opengroup.org/onlinepubs/9799919799/>.

4.2 Informative References

- **[IEC 61131-3]** IEC 61131-3, "Programmable controllers — Part 3: Programming languages". Its on-delay, off-delay, and pulse timers correspond to the timers of Section 10.5.
- **[OAP-AN]** Open Aquarium Project, "Open Aquarium Alarms and Notifications", Revision 0.3, [specs/alarms-and-notifications/spec.md](#).
- **[OAP-WL]** Open Aquarium Project, "Open Aquarium Water Level and Water Change", Revision 0.3, [specs/water-level-and-water-change/spec.md](#).
- **[OAP-DO]** Open Aquarium Project, "Open Aquarium Dosing", Revision 0.3, [specs/dosing/spec.md](#).
- **[OAP-WT]** Open Aquarium Project, "Open Aquarium Water Testing", Revision 0.3, [specs/water-testing/spec.md](#).
- **[OAP-FP]** Open Aquarium Project, "Open Aquarium Flow Programs", Revision 0.3, [specs/flow-programs/spec.md](#).
- **[OAP-PI]** Open Aquarium Project, "Open Aquarium Probe Interface", Revision 0.4, [specs/probe-interface/spec.md](#).
- **[OAP-MM]** Open Aquarium Project, "Open Aquarium Matter Mapping", Revision 0.3, [specs/matter-mapping/spec.md](#).
- **[OAP-CA]** Open Aquarium Project, "Open Aquarium Controller API", Revision 0.1, [specs/controller-api/spec.md](#).

5 Terms, Definitions, and Conventions

5.1 Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC 2119] [RFC 8174] when, and only when, they appear in all capitals, as shown here.

5.2 Terms and Definitions

Terms defined in the Device Classes specification [OAP-DC], such as device, endpoint, class, feature, controller, and revert target, and in the Fail-Safe Behavior specification [OAP-FS], such as fail-safe behavior, inactive state, control function, unavailable input, lost device, stop-on-loss equipment, safety function, and radio link, have the same meaning here.

Automation: The part of a controller that decides what its outputs do.

Configuration: An owner's complete automation for one controller, written as a configuration document (Section 7).

Element: One named part of a configuration: an output, input, condition, control function, interlock, mode, or step program.

Output: Something a configuration controls, usually bound to an output or unit endpoint (Section 8).

Virtual output: An output that is not bound to an endpoint, whose state other elements can use (Section 8.4).

Input: A named value a configuration reads, such as a sensor reading, an output's state, or a value derived from other inputs (Section 9).

Condition: A named value that is true, false, or unknown, such as a comparison, a schedule, or a combination of other conditions (Section 10).

Unknown: The value of a condition, or of an input, that cannot be determined, for example because an input it uses is unavailable.

Running: The state of an output that is doing its job: on, open, enabled, or at a level above 0 %.

Stopped: The state of an output in its inactive state (Fail-Safe Behavior Table 2).

Demand: What one part of a configuration wants an output to do in one evaluation: run, stop, a level, or the output's fail-safe behavior.

Layer: One level of precedence among the demands on an output, such as interlocks, overrides, or the output's control function (Table 25).

Resolved state: What an output does after every demand on it has been weighed in order of precedence (Section 17.3).

Effective setpoint: The setpoint a function uses at a given moment, after schedules, step programs, offsets, and limits (Section 12).

Hysteresis: The difference between the value at which an output switches on and the value at which it switches off.

Dead band: A range of values in which neither of two opposing outputs, such as a heater and a chiller, runs.

Interlock: A rule that keeps outputs stopped while a condition holds (Section 13).

Mode: A state of the whole system, such as feeding or maintenance, that changes what several outputs do (Section 14).

Override: An owner's manual setting of one output's state, which expires (Section 15).

Step program: A sequence of steps, each with setpoints and a duration, for a temporary regime such as a quarantine treatment (Section 16).

Program run: One use of a step program, from the start time the owner gives until the owner stops it.

Preset: A recommended set of elements for one kind of equipment (Section 18).

Binding: The link from an output or input of a configuration to an endpoint of a particular device (Section 8.5).

Evaluation: One pass of a controller through its whole configuration (Section 17.1).

Temporary state: A state with a known end time, set by a mode, an override, a program step, or `stop-all` (Section 17.5).

Latched stop: A stop with no known end that an endpoint keeps through loss of contact and power cuts until a controller ends it [OAP-DC Section 7.4.13; OAP-FS Section 8.4], which a controller sends for interlocks and emergency modes (Section 17.5).

Hosted function: A control function of a type that another Open Aquarium specification defines, such as a lighting or dosing function, which this Specification hosts (Section 11.10).

Extended state: The state a mode keeps an output in for a time after the mode ends (Section 14.4).

Saved state: The parts of a controller's automation that change while it runs and are kept across restarts, such as active modes and overrides (Section 17.8).

5.3 Notation

JSON member names, element types, and the names of attributes, commands, and faults appear in `monospace`. Durations are in seconds. Units are Data Model unit codes [OAP-DM]. References to other Open Aquarium specifications name them, as in "Fail-Safe Behavior Section 9.5", or use their tags from Section 4, as in "[OAP-FS Table 3]". A section number with a third part refers to a numbered item of that section: "Fail-Safe Behavior Section 10.4.3" means item 3 of Section 10.4.

6 Introduction

6.1 Purpose

Four widely used commercial controller ecosystems each have their own way to program outputs: a text language, logic gates, dependencies and combiners, or a few modes per output. None of them can move an owner's programs to another brand. An owner who changes controllers must rebuild years of tuning by hand, and often loses features, such as feed modes or redundant sensors, that the new controller does differently.

This Specification defines automation once, in a portable form. It covers what these four ecosystems do, including the features that only some of them have, and adds what none of them do: a defined order of precedence, a defined response to every failed input, and programs that resume correctly after an outage.

6.2 Design Principles

1. **Function, not hardware.** A control function refers to outputs and inputs by role. Which endpoint carries out each role is a separate binding, so a function can move to another output, module, or controller.
2. **Declarative.** A configuration describes what equipment should do. It contains no program code, so it can be checked before it runs and read by any product.
3. **One decision per output, in a fixed order.** Each output has at most one control function. Interlocks, modes, overrides, and programs act on it in a fixed order of precedence, so the result never depends on the order in which elements were written.
4. **Every failure has a defined outcome.** Every input can become unavailable, and every element defines what happens then.
5. **Interlocks only stop.** Rules that protect equipment and livestock can stop outputs but never start them, because stopping is always allowed (Fail-Safe Behavior Section 8.2).
6. **Temporary states end by themselves, and stops without an end last.** A state with an end time is sent to the device with `revertAfter`, so it ends even if the controller does not. A stop with no known end, from an interlock or an emergency mode, is latched, so the device keeps it even if the controller fails.
7. **Resume from the clock.** Step programs are defined against their start time, so they resume exactly after any outage, provided the clock is valid.

6.3 Model Overview

A configuration contains outputs, inputs, conditions, control functions, interlocks, modes, and step programs. Inputs and conditions feed control functions. Each function sets one or more outputs. Interlocks, modes, overrides, and step programs also act on outputs, in a fixed order of precedence. Bindings link outputs and inputs to device endpoints.

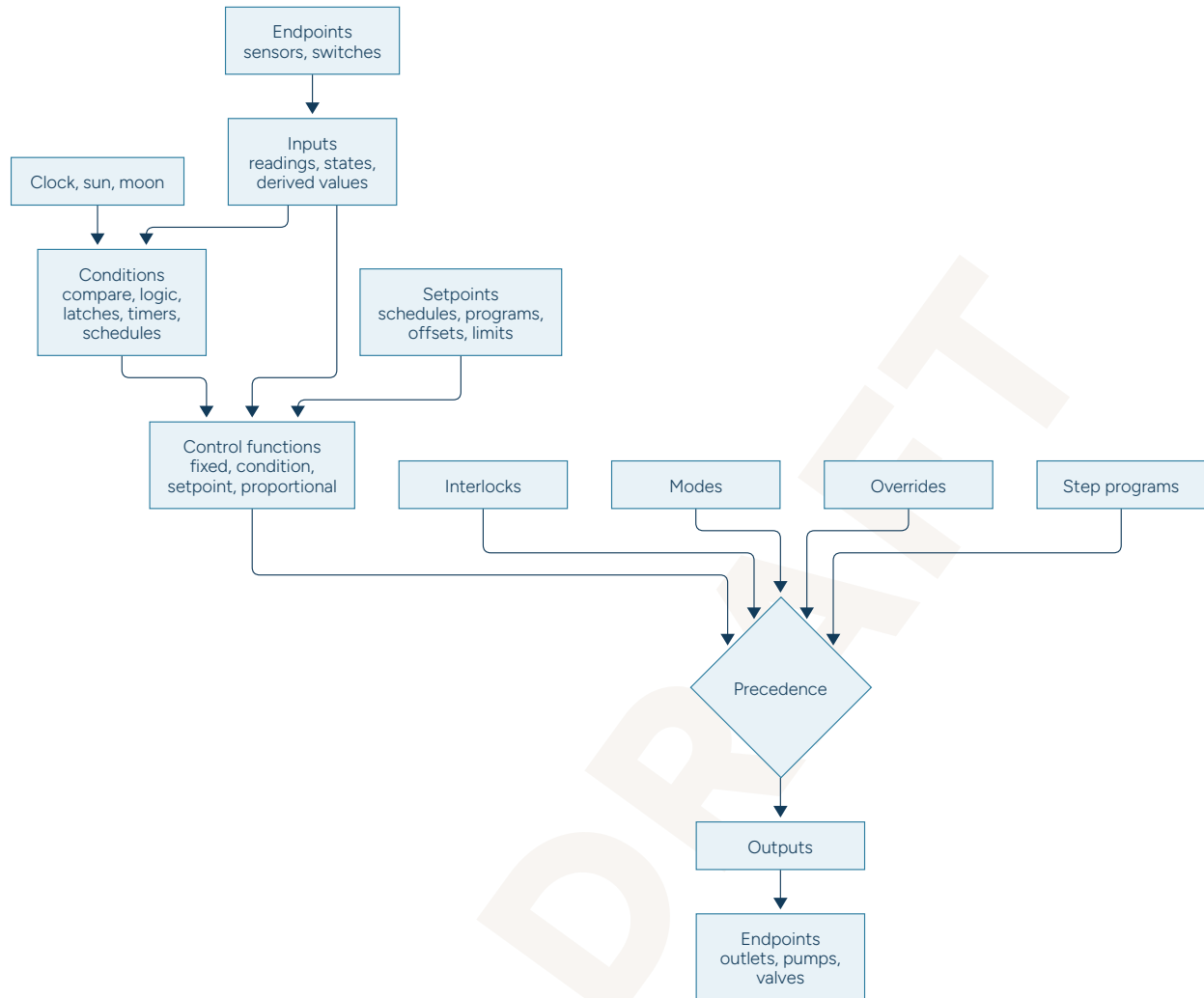


Figure 1 — Elements of a configuration

Table 1 — Kinds of element

Element	What it does	Section
Output	Something the configuration controls, bound to an endpoint or virtual.	8
Input	A value read from an endpoint, an output, the controller's history, or other inputs.	9
Condition	A value that is true, false, or unknown, computed from inputs, other conditions, the clock, or modes.	10
Control function	Sets outputs from inputs and conditions.	11
Interlock	Keeps outputs stopped while a condition holds.	13
Mode	Changes what several outputs do, for a time or until the owner ends it.	14
Override	The owner's manual setting of one output, which expires.	15
Step program	Changes setpoints and outputs over days or weeks, step by step.	16

6.4 Example

A heater controlled by the higher of two temperature sensors, as in Fail-Safe Behavior Section 10.4:

```

{
  "oapAutomation": "0.3",
  "timeZone": "Australia/Brisbane",
  "systems": [{ "id": "reef", "name": "Reef", "water": "marine", "vessels": [{ "id": "display", "role": "display" }] }],
  "equipment": [{ "id": "heater-1", "system": "reef", "category": "heater" }],
  "outputs": [{ "id": "heater", "equipment": "heater-1", "class": "switch" }],
}

```

```

"inputs": [
  { "id": "temp-a", "type": "endpoint", "class": "sensor", "parameter": "temperature", "unit": "degC" },
  { "id": "temp-b", "type": "endpoint", "class": "sensor", "parameter": "temperature", "unit": "degC" },
  { "id": "temp", "type": "maximum", "inputs": ["temp-a", "temp-b"], "minAvailable": 1, "maxDifference": 1.0 }
],
"functions": [
  { "id": "heating", "type": "setpoint", "input": "temp", "setpoint": { "value": 25.5 },
    "hysteresis": 0.4, "raise": ["heater"], "onUnavailable": "fail-safe" }
]
}

```

The heater switches on below 25.3 °C and off above 25.7 °C, acting on the higher of the two sensors. If the sensors differ by more than 1 °C, the controller raises `inputs-disagree`. If one sensor is unavailable, the heater continues on the other, and the controller raises `input-unavailable`. If both are unavailable, the heater takes its fail-safe behavior.

7 Configuration Documents

7.1 Encoding

1. A configuration document MUST be a JSON text [RFC 8259] encoded in UTF-8, and its top-level value MUST be an object.
2. Member names are case-sensitive.
3. Numbers MUST be finite.
4. A configuration document MUST be valid against the JSON Schema `schema/automation.schema.json` [JSON Schema] and meet every requirement of this Specification.

7.2 Top-Level Members

Table 2 — Top-level members

Member	Type	Presence	Description
<code>oapAutomation</code>	string	REQUIRED	Revision of this Specification. For this revision, "0.3" (Section 7.8).
<code>generator</code>	object	OPTIONAL	The product that wrote the document, as a Data Model Generator.
<code>created</code>	timestamp	OPTIONAL	When the document was written.
<code>name</code>	string	OPTIONAL	Name shown to the owner.
<code>notes</code>	string	OPTIONAL	Free text.
<code>timeZone</code>	string	REQUIRED	The time zone for local times, as a name from the Time Zone Database [TZDB], such as <code>Australia/Brisbane</code> or <code>UTC</code> .
<code>posixTimeZone</code>	string	OPTIONAL	The same time zone as a POSIX TZ string [POSIX], such as <code>AEST-10</code> , for controllers that do not carry the Time Zone Database.
<code>astronomy</code>	object	OPTIONAL	Where sun and moon times come from (Section 10.7).
<code>systems</code>	array	REQUIRED	The Data Model systems the configuration serves (Section 7.4). At least one.
<code>equipment</code>	array	OPTIONAL	The Data Model equipment the configuration refers to.
<code>outputs</code>	array	OPTIONAL	Outputs (Section 8).
<code>inputs</code>	array	OPTIONAL	Inputs (Section 9).
<code>conditions</code>	array	OPTIONAL	Conditions (Section 10).
<code>functions</code>	array	OPTIONAL	Control functions (Section 11).
<code>interlocks</code>	array	OPTIONAL	Interlocks (Section 13).
<code>modes</code>	array	OPTIONAL	Modes (Section 14).
<code>programs</code>	array	OPTIONAL	Step programs (Section 16).
<code>bindings</code>	object	OPTIONAL	Endpoint bindings, keyed by output or input identifier (Section 8.5).
<code>state</code>	object	OPTIONAL	Saved state (Section 17.8).
<code>extensions</code>	object	OPTIONAL	Extensions (Section 7.7).

When both `timeZone` and `posixTimeZone` are present, they MUST give the same local time for every date from the document's `created` date, or, in a document without `created`, from the date a controller reads it, to at least one year later. A controller that carries the Time Zone Database MUST use `timeZone`.

7.3 Identifiers and References

1. Every element has an `id`, a Data Model Identifier: 1 to 128 characters drawn from `A-Z`, `a-z`, `0-9`, `.`, `_`, `:`, and `-`.
2. Every `id` in a document MUST be unique across all elements, systems, vessels, and equipment.
3. A reference, such as a function's `input`, contains the `id` of another element, which MUST be present in the same document and MUST be of the kind the reference requires.
4. Producers SHOULD keep identifiers the same across exports, so that a controller importing a configuration again can recognize what it already has.
5. Every element MAY have a `name`, shown to the owner, and `notes`, free text.

7.4 Systems and Equipment

`systems` and `equipment` contain Data Model System and Equipment objects [OAP-DM Sections 8.1 and 8.3], with every Data Model requirement that applies to them. They give each output and input its meaning: which system it serves, in which vessel, and which kind of equipment it is. A controller that also keeps a Data Model history SHOULD use the same identifiers in both, so that observations and events recorded by the controller refer to the same systems and equipment.

A configuration document carries no Data Model revision of its own. A controller MUST read its systems and equipment as objects of the Data Model revision that the controller implements, except those of a document whose `oapAutomation` is `"0.1"`, which it MUST first upgrade as Data Model Section 7.7.6 requires, so that each reservoir without `circulating` is circulating [OAP-DM Section 7.7]. A producer SHOULD write `circulating` for every reservoir, so that a document means the same under every revision of the Data Model.

7.5 Numbers and Units

1. Every number that describes a value of an input, such as a setpoint, threshold, hysteresis, or limit, is in that input's unit (Section 9.1). A controller converts readings to the input's unit with the Data Model registry [OAP-DM Section 9.5], never the other way round, so a configuration means the same thing on every controller.
2. Some numbers are differences rather than values: hysteresis, dead bands, proportional bands, offsets, and maximum differences. A program that converts a configuration to another unit MUST convert a difference with the factor alone, without the offset. For example, a hysteresis of `0.5 degC` is `0.9 degF`.
3. Levels are in percent, from 0 to 100.
4. Durations are in seconds.

7.6 Times, Dates, and Durations

1. A timestamp is a Data Model Timestamp: an RFC 3339 date and time with a UTC offset [RFC 3339] [OAP-DM Section 7.4].
2. A date is a Data Model Date, such as `2026-10-14`.
3. A month and day, such as `12-01`, recurs every year. It MUST be a day that exists in every year, so `02-29` is not allowed.
4. A time of day, such as `08:30` or `08:30:15`, is a local time in the configuration's time zone.
5. Days of the week are `mon`, `tue`, `wed`, `thu`, `fri`, `sat`, and `sun`.
6. A duration is a number of seconds, and is measured as elapsed time, whatever the local clock does.
7. When a local time does not exist on a date because the clocks go forward, it means the first moment after the gap. When a local time occurs twice because the clocks go back, it means the first occurrence.

7.7 Extensions

Extensions follow Section 7.6 of the Data Model.

1. The top-level object, `astronomy`, every element, and the saved state MAY contain an `extensions` member, whose members are extension names.

2. The `type` of an input, condition, function, or interlock, and the `kind` of a mode, MAY be an extension name. An element of an extension type has an `id`, and MAY have a `name`, `notes`, and further members, which the party that owns the extension name defines.
3. A controller that does not support an element of an extension type MUST treat it as unsupported (Section 19.2).
4. Producers MUST NOT add members that this Specification does not define, except inside `extensions` and in elements of extension types.
5. Consumers that write documents they have read SHOULD preserve extensions they do not understand.

7.8 Revisions and Compatibility

Before revision 1.0, any revision of this Specification may make incompatible changes. From revision 1.0 onward, compatibility follows the same rules as the Data Model [OAP-DM Section 7.7].

A controller of this revision MUST accept a document whose `oapAutomation` is "0.1" or "0.2", and MUST read it as a document of this revision, except that:

1. it leaves out an override's `by` that is not a UUID, in a document of revision 0.1, which allowed any string there (Table 23);
2. it reads the systems and equipment of a document of revision 0.1 as Section 7.4 says; and
3. it reads a function of a provisional extension type that Table 14 replaced, and its saved state under `state.extensions`, as the specification that defines the function says, which is as a function of the type that replaced it, with its saved state in `state.functions` (Section 11.10).

Otherwise such a document MUST meet every requirement of this revision, so one that breaks a rule that revision 0.2 or 0.3 added, such as a mode of an extension kind without an end (Section 14.1), or a mode command `prime` (Section 14.5), is not valid. A controller that exports a document writes "0.3", with the function types and saved state of this revision.

8 Outputs

8.1 Output Members

Table 3 — Output members

Member	Type	Presence	Description
<code>id</code>	identifier	REQUIRED	
<code>equipment</code>	identifier	REQUIRED unless virtual	The Data Model equipment the output controls.
<code>channel</code>	identifier	OPTIONAL	A Channel of <code>equipment</code> , such as one head of a doser.
<code>class</code>	string	REQUIRED unless virtual	The Device Classes class of the endpoint that carries the output: an output or unit class, or an extension name.
<code>features</code>	array	OPTIONAL	Features the endpoint MUST implement, such as <code>thermostat</code> .
<code>virtual</code>	boolean	OPTIONAL	<code>true</code> for a virtual output (Section 8.4). Default <code>false</code> .
<code>runState</code>	object	OPTIONAL	The parameters of the class's <code>set</code> command that put the output in its running state, for example <code>{"running": true, "speed": 70}</code> . Default as in Table 5.
<code>restartDelay</code>	number (s)	OPTIONAL	After the output stops, no element may start it again for this long (Section 17.3). Default 0.

`runState` corresponds to `run` in Fail-Safe Behavior Section 13.2: the equipment's normal running state as the owner sets it. It MUST NOT be the inactive state, such as `{"on": false}`. A virtual output MUST NOT have `class`, `features`, `channel`, or `runState`.

8.2 Output States

Every element that acts on an output asks for one of the states in Table 4.

Table 4 — Output states

State	JSON	Meaning
Run	"run"	The output's running state: its <code>runState</code> .
Stop	"stop"	The output's inactive state (Fail-Safe Behavior Table 2).
Level	{"level": 40}	A level from 0 to 100 %, for outputs of the classes in Table 5 that have one. A level of 0 is the same as stop.
Fail-safe	(no JSON)	The output's fail-safe behavior, as if contact had been lost. Only a function's unavailable-input behavior asks for it (Section 11.9).

8.3 Output Classes

Table 5 — How automation controls each class

Class	Running state (default <code>runState</code>)	Stopped by	Running when the endpoint reports	Level
switch	{"on": true}	set on false	on	—
level-output	{"level": 100}	set level 0	level above 0	level
pump	{"running": true}	set running false	running	speed, with running true above 0
light	{"on": true}	set on false	on	—
heater	{"enabled": true}	set enabled false	enabled	—
chiller	{"enabled": true}	set enabled false	enabled	—
valve	{"open": true}	set open false	open	—
auto-top-off	{"enabled": true}	set enabled false	enabled	—
analyzer	{"enabled": true}	set enabled false	enabled; for an analyzer of Device Classes revision 0.2 or earlier, state other than disabled	—
doser	none	stop; with the <code>local-program</code> feature, also set enabled false	dosing	—
feeder	none	stop	feeding	—
filter-roller	none	stop	state advancing	—

1. An output of a class with no running state can only be stopped, or left to the function that drives it. Doses, feedings, and advances are operations rather than states [OAP-DC Section 7.14]: dosing functions [OAP-DO] and water changes [OAP-WL] send `dose` to the dosing heads they drive, and mode commands (Section 14.5) and the classes' own programs start the others. While such an output's resolved state is stop, nothing starts a new operation on it: a dosing function or a water change sends no `dose` to a head that a higher layer holds stopped. A dosing head that runs its own program is held while it is stopped, so that its program starts no dose either [OAP-DC Section 7.13].
2. Lights and pumps that run their own programs (the `local-program` and `wave` features) are usually driven by lighting functions [OAP-LP] and flow functions [OAP-FP], whose demand on each light or pump is its program. Automation stops them, sets them to a level, or leaves them to their program. When the demands of every higher layer on a light or a pump end, its function returns it to its program, as those specifications define [OAP-LP Section 7.6; OAP-FP Section 7.6].
3. A heater or chiller with the `thermostat` feature regulates by itself while it is running. A `device-setpoint` function can give it its setpoint (Section 11.7).

8.4 Virtual Outputs

1. A virtual output has a state, computed like that of any other output, but no endpoint.
2. Its state is available to other elements through an `output` input (Section 9.3).
3. Functions, modes, overrides, programs, and interlocks act on virtual outputs as on any other output.
4. Its fail-safe behavior is stop.

Virtual outputs let several functions follow one shared state that the owner can see and override. For example, a virtual output `night`, driven by a schedule, can slow the return pump and enable a moonlight; overriding it tests the whole night behavior at once. A virtual output can later be bound to real equipment without changing the elements that use it.

8.5 Bindings

1. `bindings` links each output and endpoint input (Section 9.2) to an endpoint, with a `device` and an `endpoint`.
2. `endpoint` is the endpoint's `id` [OAP-DC Section 7.1].
3. `device` identifies the device as `<binding>:<identifier>`, where the binding defines the identifier:
 - `module-bus`: followed by the module's 128-bit unique ID [OAP-MB Section 9.1] as 32 lowercase hexadecimal digits;
 - `controller:self` for the controller's own built-in endpoints;
 - `device-protocol`: followed by the device's key ID, or, for a device behind an adapter, the adapter's key ID, a slash, and the device's identifier on the adapter, as Device Protocol Section 7.3 defines; and
 - an extension name followed by a colon, for other bindings. A `device` of any other form is invalid.
4. A binding MAY have a `description`, shown to the owner, such as "Power strip, outlet 1".
5. A virtual output MUST NOT be bound.
6. The endpoint bound to an output MUST implement the output's `class` and `features`. The endpoint bound to an input MUST implement the input's `class`, and, where the input names a `parameter`, measure it. A controller MUST check this whenever it binds an endpoint or reads a device description, and MUST NOT use a binding that fails it.
7. An output with no usable binding is never commanded, and an input with no usable binding is unavailable. The controller raises `binding-missing` for each (Section 17.9).

Bindings are the only part of a configuration that depends on the hardware. A configuration without bindings can be shared with other owners (Section 19.3).

9 Inputs

9.1 Input Types

Every input has an `id` and a `type`. An input is a number in a unit, true or false, or one of a list of values, as its type and source determine. A number's unit is the unit given by the input, or, where it gives none, the unit of the attribute it reads.

Table 6 — Input types

Type	Value	Section
<code>endpoint</code>	An attribute of a bound endpoint, such as a sensor's reading or a level switch's <code>wet</code> .	9.2
<code>output</code>	Whether an output is running, or an attribute of the endpoint that carries it, such as its current.	9.3
<code>fault</code>	Whether a fault is active on an endpoint or its device.	9.4
<code>observation</code>	The latest Data Model observation of a parameter, such as the owner's last alkalinity test.	9.5
<code>moon-illumination</code>	The illuminated fraction of the moon, in %.	9.6
<code>average, minimum, maximum</code>	A value derived from several inputs.	9.7

9.2 Endpoint Inputs

Table 7 — Endpoint input members

Member	Type	Presence	Description
<code>class</code>	string	REQUIRED	The Device Classes class of the endpoint.
<code>attribute</code>	string	see Table 8	The attribute read.

Member	Type	Presence	Description
parameter	string	see Table 8	The Data Model parameter measured, for sensor, analyzer, and scaled analog-input endpoints.
unit	string	see Table 8	The unit in which the configuration gives this input's values. MUST be accepted for parameter, or convertible from the attribute's unit.
equipment	identifier	OPTIONAL	The Data Model equipment of the sensor, such as a probe.
channel	identifier	OPTIONAL	A Channel of equipment.
system, vessel	identifier	OPTIONAL	Where the value is measured.
maxAge	number (s)	OPTIONAL	The oldest reading the input accepts (Section 9.8).

Table 8 — What an endpoint input reads

Class	Without attribute, reads	Also REQUIRED
sensor	value, a number	parameter, unit
analyzer	The latest result for parameter in lastResults, a number; for an analyzer that a testing function drives, that function's latest accepted result, corrected (item 3)	parameter, unit
analog-input with parameter	value, a number	unit
analog-input without parameter	voltage, in V	—
level-switch	wet, true or false	—
leak-sensor	leak, true or false	—
contact-input	closed, true or false	—
any other class	—	attribute

1. An input that names an attribute reads that attribute of the class, its features, or the common endpoint and output attributes [OAP-DC Section 8]. Boolean attributes are true or false, enum attributes are one of their values, and number attributes are numbers. Other attribute types cannot be inputs.
2. A reading whose qualifier is < or > [OAP-DC Section 11.1] is a limit, not a value. A controller MUST treat it as the limit for comparisons that it decides, for example a reading of > 14 as above 13, and as unavailable for anything else, such as a setpoint function or an average. This applies to a qualified reading whose quality is good, such as one that a sensor reports beyond its measuring range while it can tell that its probe works; a reading of any other quality is unavailable (Section 9.8).
3. An input of the class analyzer bound to the endpoint of an analyzer that a testing function drives MUST read that function's latest accepted result of its parameter, corrected, with the result's sample time as the reading's time, rather than lastResults as it is, as the Water Testing specification defines [OAP-WT Section 13.1]. So a function never acts on a result that the testing function rejected, or holds back as suspect.

9.3 Output Inputs

An output input reads the state of an output.

1. Without attribute, it is true while the output is running. For an output bound to an endpoint, this is what the endpoint reports, as in Table 5. For a virtual output, it is the output's resolved state in the same evaluation.
2. With attribute, it reads that attribute of the output's endpoint, such as current or power, as in Section 9.2. For a virtual output, the only attribute is level, its level in %.
3. While the device of an output's endpoint is lost, an output input with attribute is unavailable, as every input of a lost device is (Fail-Safe Behavior Section 10.1). Without attribute, the output's state is unknown, and the input unavailable, until the time of Fail-Safe Behavior Section 7.6.3 has passed. After that, a controller takes the output to be in its fail-safe behavior, as Fail-Safe Behavior Section 7.6.3 requires:
 - stopped, while a latched stop that the controller sent holds it (Section 17.5) [OAP-FS Section 8.4];

- otherwise, with `failSafeAction` state, in its `failSafeState`; and
- with `continue`, in the state the controller last commanded, or, from the time a revert that was pending then falls due, in that revert's target, because the endpoint keeps its pending revert [OAP-FS Section 7.3.2], unless the endpoint follows a local program, whose state stays unknown until the device reports again.

For example, a pump paused for feeding with `revertAfter`, whose device is then lost, counts as stopped until the pause would have ended, and as running after that.

Because an output input reads what a bound endpoint reports, it never depends on the evaluation in progress, and can be used by the function that drives the same output.

9.4 Fault Inputs

A `fault` input is true while a fault is active on the endpoint of the output or endpoint input named by `of`.

1. `codes` lists Device Classes fault codes or extension names; without it, any fault counts.
2. With `device` `true`, the input reads the device's faults instead of the endpoint's.
3. The input is unknown while the device is lost (Fail-Safe Behavior Section 7.6).

For example, a fault input on a heater's outlet with `codes` `["stuck-on"]` detects a welded relay.

9.5 Observation Inputs

An `observation` input reads the latest Data Model observation of `parameter` for `system`, and optionally `vessel`, from the controller's history.

1. `unit` and `maxAge` are REQUIRED. An observation older than `maxAge` is unavailable.
2. `methods` MAY limit the observations to Data Model method types, such as `titration`. A method type does not tell instruments apart: an analyzer's titration and a person's are both `titration` [OAP-DM Table 25], so an input that should follow one analyzer is an `endpoint` input (Section 9.2).
3. Observations with a `qualifier` are treated as in Section 9.2.
4. The input MUST pass over the observations that the Data Model says not to act on: one whose `unitAssumed` is `true`, until a person has confirmed its basis [OAP-DM Section 9.4.4], and one whose time is later than the controller's clock, until that time has passed [OAP-DM Section 7.4.2]. The latest observation is then the latest of the others.
5. Results of analyzers are observations only once their testing function has accepted them: suspect and rejected results are not observations [OAP-WT Section 13.1.2].

Observation inputs let functions use manual tests and laboratory results, for example to stop CO2 while the owner's last carbonate hardness test was low. How analyzer results become observations is defined by the Water Testing specification [OAP-WT].

9.6 Moon Illumination

A `moon-illumination` input is the fraction of the moon's disc lit by the sun, in %, at the current time. It needs `astronomy.location` (Section 10.7), and is unavailable while the clock is not valid.

9.7 Derived Inputs

An `average`, `minimum`, or `maximum` input combines two or more numeric `inputs` of the same parameter and unit.

Table 9 — Derived input members

Member	Type	Presence	Description
<code>inputs</code>	array	REQUIRED	Two or more inputs.
<code>minAvailable</code>	integer	OPTIONAL	How many inputs must be available for the derived input to be available. Default: all of them.
<code>maxDifference</code>	number	OPTIONAL	The largest difference allowed between available inputs.
<code>onDisagreement</code>	string	OPTIONAL	<code>continue</code> (default) or <code>unavailable</code> : what the derived input does while its inputs differ by more than <code>maxDifference</code> .

1. The value is the average, minimum, or maximum of the inputs that are available.

2. While fewer than `minAvailable` inputs are available, the derived input is unavailable.
3. While some, but at least `minAvailable`, inputs are available, the derived input continues with them, and the controller MUST raise `input-unavailable` for each unavailable one (Fail-Safe Behavior Section 10.4.3).
4. While the largest and smallest available inputs differ by more than `maxDifference`, the controller MUST raise `inputs-disagree`, and the derived input continues or becomes unavailable as `onDisagreement` says.

The RECOMMENDED method for heater control in Fail-Safe Behavior Section 10.4.2 is a `maximum` of two temperature inputs, with `minAvailable` 1 and `maxDifference` 1 °C.

9.8 Availability and Maximum Age

An input is unavailable in each case of Fail-Safe Behavior Section 10.1, and also when:

1. it has no usable binding (Section 8.5);
2. its value is older than its `maxAge`, measured as Fail-Safe Behavior Section 10.2.2 requires;
3. it needs the time, sun, or moon, and the clock is not valid (Fail-Safe Behavior Section 10.5);
4. its type, class, or attribute is not supported by the controller;
5. it reads an output whose state is unknown (Section 9.3.3); or
6. it reads an endpoint of a probe whose inputs the Probe Interface specification holds: during a calibration and until a person confirms that the probe is back [OAP-PI Section 14.8.1], after a fault or a reading that was not `good` until its readings have settled [OAP-PI Section 14.8.6], after a person makes another endpoint of the probe's port active [OAP-PI Section 7.1.5], and after a smart probe is replaced, until a person confirms the new probe [OAP-PI Section 16.4.2].

Table 10 — Default maximum ages

Input	Default <code>maxAge</code>
endpoint from a sensor or scaled analog-input	The longest of 60 s, three times the sensor's <code>measurementInterval</code> , where it reports one, and three times the reporting interval of its binding (Fail-Safe Behavior Section 10.2.3)
endpoint from an analyzer	48 h (Fail-Safe Behavior Section 10.2.3)
Any other endpoint input, and an output input of an output bound to an endpoint	The longer of 60 s and three times the reporting interval of its binding, from the last time the controller received the attribute
observation	none: <code>maxAge</code> is REQUIRED

7. A controller MUST check the age of every reading it uses, against the input's `maxAge` or the default in Table 10.
8. The reporting interval of a binding is the longest time it lets the endpoint go without reporting its state: 1 s for an input endpoint on the Module Bus, and 5 s for any other endpoint there [OAP-MB Section 11.1]; and, over the Device Protocol, the `maxInterval` the device returned [OAP-DP Section 11.5].
9. A controller MUST warn the owner when an input's `maxAge` is shorter than three times the reporting interval of its binding, or, for a sensor, three times its `measurementInterval`, because the input would then become unavailable while its endpoint works normally.

For example, a temperature sensor on a battery-powered network device that reports every 600 s, and measures every 300 s, has a default `maxAge` of 1800 s, and a configured `maxAge` of 120 s makes the controller warn.

10 Conditions

10.1 Values and Unknown Values

A condition is true, false, or unknown. It is unknown when it cannot be determined, for example because an input it compares is unavailable. An unknown condition counts as unavailable wherever this Specification or the Fail-Safe Behavior specification refers to unavailable inputs. Wherever a condition is expected, a reference MAY also name an input that is true or false.

Table 11 — Condition types

Type	True when	Section
compare	An input is above or below a threshold, or equal to a value.	10.2
all, any, not	All, any, or none of its conditions are true.	10.3
latch	It has been set, and not reset since.	10.4
timer	Its input has been true for a time, or was true recently.	10.5
schedule	The time is within one of its windows.	10.6
mode	A mode is active.	10.8
unavailable	An input or condition is unavailable.	10.8

10.2 Comparisons

A `compare` condition compares its `input` with exactly one of `above`, `below`, or `equals`.

1. With `above` T, it is true when the input is above T. With `hysteresis` h, once true it stays true while the input is above T - h.
2. With `below` T, it is true when the input is below T. With `hysteresis` h, once true it stays true while the input is below T + h.
3. With `equals`, it is true when an input that is true or false, or one of a list of values, equals the given value.
4. It is unknown while its input is unavailable.
5. When it becomes known, after it has been unknown or after the controller starts, its hysteresis starts afresh, as if it had been false: with `above` T, it is true only above T, and with `below` T, only below T. This affects only its hysteresis. A change from unknown to true is not a change from false to true, so it starts no mode (Section 14.3.1) and no `pulse` (Section 10.5.3).

For example, `{"type": "compare", "input": "ph", "above": 8.45, "hysteresis": 0.05}` becomes true above pH 8.45 and false again at or below pH 8.40.

10.3 All, Any, and Not

`all` and `any` combine the conditions listed in `of`; `not` inverts the one condition named by `of`. Unknown values combine as in Table 12, so that a result is known whenever the known values decide it.

Table 12 — Combining unknown values

A	B	all of A and B	any of A and B	not A
true	true	true	true	false
true	false	false	true	false
true	unknown	unknown	true	false
false	false	false	false	true
false	unknown	false	unknown	true
unknown	unknown	unknown	unknown	unknown

10.4 Latches

A `latch` remembers that its `set` condition has been true.

1. When `set` is true, the latch is set.
2. When `set` is false, and `reset` is true or the owner resets it, the latch is cleared. Because `set` wins, a latch cannot be cleared while its `set` condition is true or unknown.
3. An unknown `set` or `reset` does not change the latch.
4. `ownerReset` (default `true`) allows the owner to reset the latch. A latch without `reset` MUST allow the owner to reset it.
5. A latch is never unknown. It starts cleared, and a controller MUST keep its value across restarts (Section 17.8).

10.5 Timers

A `timer` condition follows its `input` with a delay, according to its `kind`, as the timers of IEC 61131-3 do [IEC 61131-3]:

1. `on-delay`: true once the input has been true without interruption for `duration`; false as soon as the input is not true.
2. `off-delay`: true while the input is true, and for `duration` after it stops being true.
3. `pulse`: true for `duration` from the moment the input changes from false to true, whatever the input does meanwhile. A new change during the pulse does not extend it.

While the input is unknown, the timer is unknown. When the input is known again, the timer starts afresh, and takes the input's new value as its previous value. Timers measure elapsed time. They are not kept across restarts. After a restart, each takes its input's first value as its previous value (Section 17.7), so an `on-delay` starts timing and a `pulse` does not start.

10.6 Schedules

A `schedule` condition is true during its windows, on the days it selects.

Table 13 — Schedule members

Member	Type	Presence	Description
<code>windows</code>	array	REQUIRED	One or more windows, each with a <code>start</code> , and either an <code>end</code> or a <code>duration</code> of at most 24 h.
<code>days</code>	array	OPTIONAL	Days of the week on which windows start. Default: every day.
<code>everyDays</code> , <code>from</code>	integer, date	OPTIONAL	Windows start only every <code>everyDays</code> days, counting from the date <code>from</code> . Both or neither.
<code>dates</code>	array	OPTIONAL	Ranges of dates, each with <code>from</code> and <code>to</code> , inclusive, on which windows start. Both are months and days, recurring each year, or both are dates.

`start` and `end` are each a time of day, or an object with an `event`, one of `sunrise`, `sunset`, `solar-noon`, `moonrise`, or `moonset`, and an optional `offset` in seconds.

1. A window starts on each local date that every given filter selects: `days`, `everyDays`, and `dates`. `everyDays` `N` selects `from` and every `N`th day after it. A range of months and days whose `from` is later than its `to`, such as 12-01 to 02-28, runs over the new year.
2. On a selected date, the window starts at its `start`: the time of day on that date, or the event on that date plus its `offset`.
3. It ends at its `end`: the first moment after the start with that time of day, or the first occurrence of that event after the start plus its `offset`; or at the start plus `duration`. A window that would last more than 24 h ends 24 h after its start. So a window from 21:00 to 07:00 ends the next morning.
4. The schedule is true from the start of a window until its end.
5. On a date when an event does not occur, such as a day when the moon does not rise, windows that start at that event do not start. A window that ends at an event that does not occur within 24 h ends 24 h after its start.
6. Local times follow Section 7.6, including changes to and from daylight saving time.
7. The schedule is unknown while the clock is not valid (Fail-Safe Behavior Section 10.5).

A controller MUST change a schedule's value within 1 s of the time a window starts or ends.

10.7 Sun and Moon

`astronomy` gives the source of sun and moon times, with at least one of:

- `location`: a `latitude`, from -60 to 60, and a `longitude`, from -180 to 180, in degrees (WGS 84), and an optional `name`; or
- `sunTable`: a table of local `sunrise` and `sunset` times, each for a month and day (`date`).

It MAY also have `shift`, `noonAt`, and `shiftDays`, which move the events (item 3).

1. With `location`, the controller computes the events for that place, by the method `oap-1` of the Lighting Programs specification [OAP-LP Section 9]:

- **sunrise** and **sunset**: the moments when the centre of the sun is 0.833° below the horizon, which allows for refraction and the sun's size, at sea level [OAP-LP Sections 9.2 and 9.3];
- **solar noon**: the moment the sun crosses the meridian [OAP-LP Section 9.3]; and
- **moonrise** and **moonset**: the moments when the top of the moon's disc touches the horizon, with refraction and parallax allowed for, which are the moments when the moon's altitude α [OAP-LP Section 9.4.14] is $-(0.5667^\circ + s)$, where $s = \text{asin}(1737.4 / \Delta)$ is the moon's semidiameter and Δ its distance in km [OAP-LP Section 9.4.8].

Moon illumination is the moon's illuminated fraction k [OAP-LP Section 9.4.10], in %. Latitudes are limited to 60° north and south, as the Lighting Programs specification limits them, because nearer the poles the sun may not rise or set for days [OAP-LP Section 9.1.6]; a configuration for such a place uses `sunTable`.

2. With `sunTable`, the controller interpolates sunrise and sunset linearly between the entries by day of the year, counted as in Section 12.2, wrapping from the last entry to the first, and takes solar noon as halfway between them, as the Lighting Programs specification reads a seasonal table [OAP-LP Section 9.5]. Each date MUST appear only once in the table, and each entry's `sunrise` MUST be earlier than its `sunset`. When both are given, `sunTable` is used for the sun. Moon events, and `moon-illumination`, need `location`.
3. Every event, from either source, is moved by the shift of its date, which lets a biotope's day, computed for its own location, fall at a convenient time where the aquarium is. The shift is that of Lighting Programs Section 9.3.1: with `noonAt`, a local time of day, the shift that puts the location's mean solar noon at that time on the date; otherwise `shift`, in seconds, from -86400 to 86400 , plus 86400 times `shiftDays`. `noonAt` needs `location`, and MUST NOT be given with `shift`. With `shiftDays`, a whole number of days from -366 to 366 , each date has the sun of the date that many days before it, and the moon follows it [OAP-LP Section 9.1.5]; a `sunTable` is read for the date that many days before.
4. Events are moments in time, so a location in another time zone gives its sunrise at the corresponding local time where the controller is. The sunrise, solar noon, and sunset of a local date are those of its solar day, the one whose mean noon, after the shift, is nearest local noon on that date [OAP-LP Section 9.3]; for a location in the controller's own time zone, these are the date's own. A moon event on a date is the occurrence whose moment, after the shift, falls on that date in the controller's local time. `moon-illumination` at a moment u is the illuminated fraction at the moment $u - s$, where s is the shift of the local date of u , as the Lighting Programs specification moves moonlight [OAP-LP Section 10.1.1].
5. A controller MUST compute sunrise, solar noon, and sunset within 2 s, moonrise and moonset within 60 s, and moon illumination within 0.1 percentage points, of the method `oap-1`, for latitudes from 60° S to 60° N and dates from 2000 to 2100 (provisional). So schedules and lights that follow the same sun agree to the second [OAP-LP Section 9.6].

The location need not be the owner's own. See Section 21 about privacy.

10.8 Mode and Unavailable Conditions

1. A `mode` condition is true while the mode named by `mode` is active, but not while its outputs are extended after it ends (Section 14.4). It is never unknown.
2. An `unavailable` condition is true while the input or condition named by `of` is unavailable or unknown. It is never unknown. It lets a configuration act on a failure, for example by switching on a warning light.

11 Control Functions

11.1 Common Members

A control function sets one or more outputs. Each output MUST be driven by at most one function.

Table 14 — Function types

Type	Sets its outputs	Section
<code>fixed</code>	Always to the same state.	11.2
<code>condition</code>	To one state while a condition is true, and another while it is false.	11.3

Type	Sets its outputs	Section
setpoint	On and off to keep an input near a setpoint, with hysteresis.	11.4
proportional	To a level proportional to the input's distance from a setpoint.	11.6
device-setpoint	Running, and gives self-regulating equipment its setpoint.	11.7
alarm	To signal alarms, as the Alarms and Notifications specification defines.	11.10
top-off	To keep a vessel's water level, as the Water Level and Water Change specification defines.	11.10
water-change	To change part of a system's water, as the Water Level and Water Change specification defines.	11.10
ro-di	To fill a reservoir with purified water, as the Water Level and Water Change specification defines.	11.10
dosing	To dose, as the Dosing specification defines.	11.10
water-testing	To test the water with analyzers, as the Water Testing specification defines.	11.10
lighting	To a lighting program, as the Lighting Programs specification defines.	11.10
flow	To a flow program, as the Flow Programs specification defines.	11.10

Every function MAY have:

- **preset**: the preset it was created from (Section 18);
- **timing**: timing limits (Section 11.8), for **condition** and **setpoint** functions; and
- **onUnavailable**: what it does when an input is unavailable (Section 11.9), for every type but **fixed**, which uses no inputs, and the hosted types, which define their own (Section 11.10).

A hosted function has the members that its specification defines (Section 11.10).

11.2 Fixed

A **fixed** function sets its **outputs** to **state** all the time. It is the usual function for a return pump or a UV sterilizer: `{"type": "fixed", "outputs": ["return-pump"], "state": "run"}`.

11.3 Condition

A **condition** function sets its **outputs** to **whenTrue** (default **run**) while the condition **when** is true, and to **whenFalse** (default **stop**) while it is false. It applies its unavailable-input behavior while **when** is unknown.

With a schedule, this is a timer outlet; with logic, it can express any rule. For example, a return pump at 100 % by day and 70 % at night: `{"type": "condition", "when": "night", "outputs": ["return-pump"], "whenTrue": {"level": 70}, "whenFalse": {"level": 100}}`.

11.4 Setpoint Control

A **setpoint** function switches outputs on and off to keep a numeric **input** near its effective setpoint **S** (Section 12), with **hysteresis** **h**, which **MUST** be greater than 0.

1. Outputs listed in **raise** raise the value when running, as a heater raises temperature. Outputs listed in **lower** lower it, as a chiller lowers temperature or CO2 lowers pH.
2. Each output switches on and off at the values in Table 15, and between them keeps its previous state. With no previous decision, such as after a restart, it is off.
3. With both **raise** and **lower**, the function **MUST** have a **deadBand** **d**, greater than 0 (Section 11.5).

Table 15 — Setpoint control

Outputs	Switch on when the input is	Switch off when the input is
raise , alone	below $S - h/2$	above $S + h/2$
lower , alone	above $S + h/2$	below $S - h/2$
raise , with lower	below $S - d/2 - h$	above $S - d/2$
lower , with raise	above $S + d/2 + h$	below $S + d/2$

For example, with $S = 25.5\text{ }^{\circ}\text{C}$ and $h = 0.4\text{ }^{\circ}\text{C}$, a heater alone switches on below $25.3\text{ }^{\circ}\text{C}$ and off above $25.7\text{ }^{\circ}\text{C}$.

11.5 Heating and Cooling Pairs

A setpoint function with both `raise` and `lower` outputs keeps a dead band between them: its `raise` outputs never run above $S - d/2$, and its `lower` outputs never run below $S + d/2$. Both sides use the same input and the same effective setpoint in each evaluation, so they cannot run together however the setpoint changes, through schedules, programs, or offsets.

For example, with $S = 25.5\text{ }^{\circ}\text{C}$, $d = 1.0\text{ }^{\circ}\text{C}$, and $h = 0.4\text{ }^{\circ}\text{C}$, the heater switches on below $24.6\text{ }^{\circ}\text{C}$ and off above $25.0\text{ }^{\circ}\text{C}$, and the chiller switches on above $26.4\text{ }^{\circ}\text{C}$ and off below $26.0\text{ }^{\circ}\text{C}$. Between $25.0\text{ }^{\circ}\text{C}$ and $26.0\text{ }^{\circ}\text{C}$, neither runs.

This is the RECOMMENDED way to control a heater and a chiller together. `proportional` functions with both `raise` and `lower` outputs keep a dead band the same way (Section 11.6).

11.6 Proportional Control

A `proportional` function sets each of its outputs to a level from the distance between its numeric `input` x and its effective setpoint S . Its outputs MUST have a level (Table 5), or be virtual.

Table 16 — Proportional control members

Member	Type	Presence	Description
<code>band</code>	number	REQUIRED	The distance from the setpoint at which the level reaches <code>maxLevel</code> . Greater than 0.
<code>raise</code> , <code>lower</code>	array	at least one	Outputs that raise or lower the value.
<code>deadBand</code>	number	REQUIRED with both <code>raise</code> and <code>lower</code>	As in Section 11.5.
<code>minLevel</code>	number (%)	OPTIONAL	The level at which a running output starts. Default 0.
<code>maxLevel</code>	number (%)	OPTIONAL	The highest level. Default 100.
<code>hysteresis</code>	number	OPTIONAL	How far past the setpoint a running output continues at <code>minLevel</code> before it stops. Default 0.

1. For a `lower` output the error e is $x - S'$; for a `raise` output it is $S' - x$. S' is S for a function with outputs on one side. With both sides, S' is $S + d/2 + h$ for `lower` outputs and $S - d/2 - h$ for `raise` outputs, where h is `hysteresis`, so that neither side runs within the dead band.
2. An output starts when e is above 0, and once running, stops when e is at or below $-h$.
3. While running, its level is $\text{minLevel} + (\text{maxLevel} - \text{minLevel}) \times e / \text{band}$, kept between `minLevel` and `maxLevel`.

For example, a cooling fan with $S = 26.0\text{ }^{\circ}\text{C}$, `band` $1.0\text{ }^{\circ}\text{C}$, and `minLevel` 30 % starts at 30 % just above $26.0\text{ }^{\circ}\text{C}$, reaches 65 % at $26.5\text{ }^{\circ}\text{C}$, and runs at 100 % from $27.0\text{ }^{\circ}\text{C}$.

A controller SHOULD NOT send a new level that differs from the last one it sent by less than 1 percentage point, except to reach 0, `minLevel`, or `maxLevel`.

11.7 Device Setpoints

A `device-setpoint` function gives self-regulating equipment its setpoint, and keeps it running.

1. Its `raise` and `lower` outputs MUST be `heater` or `chiller` outputs that declare the `thermostat` feature.
2. Its setpoint values are in `unit`, a temperature unit.
3. With outputs on one side only, each is given the effective setpoint S . With both `raise` and `lower`, the function MUST have a `deadBand` d : its `raise` outputs are given $S - d/2$, and its `lower` outputs $S + d/2$. The dead band SHOULD be larger than the devices' own hysteresis.
4. While its effective setpoint is available, the function's demand is run, and the controller MUST give each output its setpoint, converted to the unit of the endpoint's `setpoint` attribute, within a minute of its differing from the endpoint's value by $0.1\text{ }^{\circ}\text{C}$ or more. It SHOULD NOT give it more often than that, to spare the device's storage.
5. While its effective setpoint is unavailable (Section 12.5), the function applies its unavailable-input behavior.

6. A setpoint whose end is known is a temporary setpoint: one that an offset with an end (Section 12.3), or a step of a step program other than its last (Section 16.2), makes differ from what the setpoint would be without it. While the output's resolved state is run, the controller MUST send a temporary setpoint with the `set` command of the output's class, with `enabled true`, its `setpoint` parameter, and `revertAfter` for the seconds until the offset or the step ends [OAP-DC Sections 7.4.5 and 9.5], so that the device returns to the setpoint it had before even if the controller fails. When the temporary setpoint ends, the controller sends the setpoint that follows with `set` and no `revertAfter`, which cancels the pending revert [OAP-DC Section 7.4.7].
7. A setpoint with no known end, such as the function's own setpoint, or one that the last step of a step program sets, and any setpoint while the output's resolved state is not run, the controller writes to the endpoint's `setpoint` attribute, which has no `revertAfter`. Such a setpoint stays on the device if the controller fails.

Each device regulates with its own sensor, so a dead band between their setpoints keeps a heater and a chiller apart only while their sensors agree (Section 13.5).

A temporary setpoint that follows another, such as the next value of an offset's ramp, keeps the target of the first one's revert [OAP-DC Section 7.4.6], so the device returns to the setpoint from before the offset or the step program. A heater or chiller of Device Classes revision 0.1 has no `setpoint` parameter on `set`, so the controller writes every setpoint to its attribute, and a temporary setpoint stays on such a device if the controller fails. The heater's own `cutoffTemperature` and the setpoint's `limits` protect against a setpoint that outlasts its controller.

11.8 Timing

A function's `timing` shapes when it starts and stops each of its outputs. "Start" means a change from stopped to running, and "stop" the reverse.

Table 17 — Timing members

Member	Meaning
<code>onDelay</code>	The function starts an output only after it has wanted it running, without interruption, for this long.
<code>offDelay</code>	The function stops an output only after it has wanted it stopped, without interruption, for this long.
<code>minOnTime</code>	Once the function starts an output, it does not stop it for this long.
<code>minOffTime</code>	Once the function stops an output, it does not start it again for this long.
<code>maxOnTime</code>	When the function has kept an output running for this long without interruption, it stops it, and the controller raises <code>max-on-time</code> .
<code>restTime</code>	After <code>maxOnTime</code> , the function keeps the output stopped for this long, then resumes. Without <code>restTime</code> , it keeps the output stopped until the owner clears <code>max-on-time</code> .

1. The delays apply first, then the minimum times, then the maximum time.
2. Timing applies to the function's own demand only. It never delays a stop by an interlock, mode, or override, or a demand caused by an unavailable input (Section 11.9).
3. After a controller starts, `minOffTime` counts from the start, as if every output had just stopped.
4. An output that `maxOnTime` holds stopped, without `restTime`, stays held across restarts (Section 17.8).

For example, a top-off valve with `maxOnTime` 120 s and `restTime` 900 s runs for at most 2 minutes at a time, and rests for 15 minutes after any run that lasts that long.

11.9 Unavailable Inputs

Every control function MUST define what it does when an input it uses is unavailable, as Fail-Safe Behavior Section 10.3 requires. The inputs a function uses are its `input` or `when`, and every input and condition they depend on, unless a derived input or a logic condition still has a known value (Sections 9.7 and 10.3). A time-varying setpoint whose time is unavailable is also an unavailable input (Section 12.5).

Table 18 — Unavailable-input behaviors

onUnavailable	The function's outputs
fail-safe (default)	Take their fail-safe behavior, as if contact had been lost (Fail-Safe Behavior Section 10.3.2): with <code>failSafeAction</code> state, the controller commands <code>failSafeState</code> ; with <code>continue</code> , it sends no command, so the endpoint keeps its last commanded state or runs its local program.
stop	Stop.
run	Run.
<code>{"level": n}</code>	Are set to level n.
hold	Keep the state the function last set them to.

1. While an input is unavailable, the controller MUST raise `input-unavailable` for it (Section 17.9).
2. When the input becomes available again, the function resumes, with its timing starting afresh.
3. `run`, `{"level": n}`, and `hold` keep equipment running without the input that controls it. Controllers SHOULD warn the owner when a function uses them for equipment whose recommended fail-safe configuration is `stop` [OAP-FS Table 3].

11.10 Functions Defined by Other Specifications

Alarm outputs, top-off, water changes, RO/DI, dosing, water testing, lighting, and flow are detailed in their own specifications. Those specifications define the hosted function types of Table 14, which this Specification's model hosts:

1. such a function drives the outputs it lists in its `outputs`, and each output still has at most one function;
2. its demand is weighed with interlocks, modes, overrides, and programs as in Section 17.3;
3. it defines its own unavailable-input behavior, as the Dosing specification may, for example, return to a fixed dose (Fail-Safe Behavior Section 10.3.2), so it has no `onUnavailable`;
4. its members other than those of Section 11.1 are defined by its specification, and a configuration document that contains it MUST be valid against the schema of that specification as well as this one; and
5. a controller keeps its saved state, in the form that its specification defines, as `state.functions["<function id>"]` (Section 17.8).

Each type is defined by:

- `alarm`: Alarms and Notifications Section 15.1 [OAP-AN];
- `top-off`, `water-change`, and `ro-di`: Water Level and Water Change Sections 7, 8, 9, and 10 [OAP-WL];
- `dosing`: Dosing Section 7 [OAP-DO];
- `water-testing`: Water Testing Section 7 [OAP-WT];
- `lighting`: Lighting Programs Section 7 [OAP-LP]; and
- `flow`: Flow Programs Section 7 [OAP-FP].

Revisions 0.1 and 0.2 carried these types under provisional extension names, such as `org.openaquariumproject.dosing`, and their saved state under `state.extensions`; a controller reads those names as Section 7.8 says. A controller that does not support a hosted type treats a function of it as unsupported (Section 19.2), and finds its outputs from its `outputs`.

The hosted functions also use the rest of this Specification:

6. Some act at the precedence of interlocks: the checks that top-off, water change, and RO/DI functions apply to every run of their outputs, such as a top-off's backup high sensor, stop those outputs whatever any other layer demands, and never start them [OAP-WL Section 7.3].
7. A water change follows a water-change mode, which it names: the mode starts it, and when the mode ends before the water change finishes, the water change finishes as that specification defines [OAP-WL Section 9.4].
8. Some stops that end the runs of top-off, water change, and RO/DI functions, and holds of top-off units, are sent without `revertAfter`, as that specification requires, as exceptions to Section 17.5.1 [OAP-WL Sections 7.5 and 8.14].

9. A testing function may drive no outputs, when it only checks the results that people enter [OAP-WT Section 7.2.3]; its demand on each analyzer it drives is stop while it delays or suspends testing [OAP-WT Section 7.2].
10. A lighting or flow function's demand on each light or pump is its program (Section 8.3.2).

12 Setpoints

12.1 Setpoint Sources

A function's `setpoint` has exactly one base: a constant `value`, or a `daily`, `seasonal`, or `dated` table. The effective setpoint is computed in this order:

1. the base, at the current time;
2. replaced by a running step program's value, where a program sets the function (Section 16);
3. plus every active offset (Section 12.3); and
4. kept within `limits` (Section 12.4).

Table 19 — Setpoint members

Member	Type	Description
<code>value</code>	number	A constant setpoint.
<code>daily</code>	array	Points, each a local <code>time</code> and a <code>value</code> , repeating every day.
<code>seasonal</code>	array	Points, each a month and day (<code>date</code>) and a <code>value</code> , repeating every year.
<code>dated</code>	array	Points, each a timestamp (<code>time</code>) and a <code>value</code> , in increasing order.
<code>interpolation</code>	string	<code>linear</code> (default) or <code>step</code> , between the points of a table.
<code>fallback</code>	number	The base used while the time is unavailable (Section 12.5).
<code>offsets</code>	array	Temporary offsets (Section 12.3).
<code>limits</code>	object	<code>min</code> and <code>max</code> (Section 12.4).

12.2 Daily, Seasonal, and Dated Setpoints

1. A `daily` table repeats every day: after its last point it continues towards its first point of the next day.
2. A `seasonal` table repeats every year by day of the year, counted in a year of 365 days, with 29 February taken as 28 February. After its last point it continues towards its first point of the next year.
3. A `dated` table holds its first value before its first point and its last value after its last point.
4. With `linear` interpolation, the base changes in a straight line from one point to the next. With `step`, it changes at each point and stays the same until the next.
5. Each time or date MUST appear only once in a table, and `interpolation` MUST NOT be given without a table.

For example, a seasonal table with 26.0 °C on 01-15 and 24.5 °C on 07-15 follows the seasons of a southern-hemisphere reef, and a daily table with 25.5 °C at 08:00 and 25.0 °C at 22:00, with `step`, lowers the temperature at night.

12.3 Offsets

An offset is a temporary change to a setpoint, such as a raised temperature during a treatment.

1. Each offset has a `value`, added to the setpoint, a timestamp `from`, a timestamp `until`, and an optional `name`. `until` MUST be later than `from`.
2. With `ramp`, the offset grows from 0 to `value` over `ramp` seconds after `from`, and shrinks back to 0 over `ramp` seconds before `until`. Twice `ramp` MUST NOT be longer than the time from `from` to `until`.
3. Offsets end by themselves, so they cannot be forgotten.

12.4 Limits

`limits` gives the lowest and highest effective setpoint the owner allows.

1. `min` MUST be less than `max`.

2. A function **MUST** have `limits` when its setpoint has a `daily`, `seasonal`, or `dated` table, when it has offsets, or when a step program sets it.
3. Every value in a setpoint's tables, its `value`, its `fallback`, and every value a step program sets it to **MUST** be within its limits.
4. A controller **MUST** keep the effective setpoint within `limits`, and **SHOULD** show the owner when it is held at a limit.

Limits are the owner's statement of what is safe for their livestock. They protect against a mistyped table, an offset added to a program, or a program meant for another tank.

12.5 When the Time Is Unavailable

While the clock is not valid (Fail-Safe Behavior Section 10.5), a `daily`, `seasonal`, or `dated` base is unavailable, and so are offsets.

1. While a step program sets the function, the program holds its last values (Section 16.4), and the function continues with them.
2. Otherwise, with a `fallback`, the function uses it as the base, without offsets, and continues.
3. Otherwise, the setpoint is an unavailable input, and the function applies its unavailable-input behavior.

A `fallback` is **RECOMMENDED** for heaters whose setpoint changes over time, so that losing the clock does not stop heating.

13 Interlocks

13.1 General

An interlock keeps outputs stopped while something is true.

1. Interlocks only stop. They never start an output or set a level.
2. Interlocks take precedence over every other element, including overrides, and share the highest layer with `stop-all` (Section 17.3).
3. An output held stopped by an interlock stops at once: no timing, minimum time, or delay holds it running.
4. The controller **MUST** show the owner which interlock holds each output stopped (Section 17.10).

Table 20 — Interlock types

Type	Holds its outputs stopped	Section
<code>block</code>	While a condition is true.	13.2
<code>requires</code>	Unless another output is running.	13.3
<code>exclusive</code>	All but one group of them, so that only one group runs at a time.	13.4

13.2 Block

A `block` interlock holds its `outputs` stopped while the condition `when` is true.

1. While `when` is unknown, it holds them stopped if `onUnavailable` is `block` (default), and allows them if it is `allow`.
2. With `latching true`, once the interlock has held its outputs stopped, it keeps holding them until the owner clears it, which is possible only while `when` is false. The controller raises `interlock-latched` meanwhile, and keeps the latch across restarts (Section 17.8).

For example, `{"type": "block", "outputs": ["kalk"], "when": "ph-high"}` stops a kalkwasser reactor while the pH is high, and also while the pH reading is unavailable.

13.3 Requires

A `requires` interlock holds its `outputs` stopped unless the output named by `requires` is running.

1. The required output is running when its resolved state in the same evaluation is running and, for an output bound to an endpoint, its endpoint also reports it running. So a dependent output stops in the same

evaluation as the output it requires, and starts only after that output has actually started. A required output whose state is unknown (Section 9.3.3) is not running.

2. With `deLay`, the required output must have been running, as in item 1, without interruption for `deLay` seconds.
3. An output MUST NOT require itself.
4. While the required output is stopped by a temporary state, such as a feed mode's pause, the interlock's stop has a known end, and is sent with `revertAfter` rather than latched (Section 17.5.5).

For example, a protein skimmer that requires the return pump with `deLay` 60 starts a minute after the return pump, and stops with it.

13.4 Exclusive

An `exclusive` interlock lets the outputs of at most one of its groups run at a time.

1. Its groups are given by `outputs`, a list of two or more outputs, each of which is a group of its own, or by `groups`, a list of two or more groups, each a list of outputs. Outputs of one group may run together.
2. Among its groups that have an output that would run after every other layer (Section 17.3), the group that had an output running at the end of the previous evaluation keeps its outputs running. If none had, the first such group in the list does. The outputs of the other groups are held stopped.
3. While the state of one of its outputs is unknown (Section 9.3.3), the outputs of every other group are held stopped, because that output may be running, as Fail-Safe Behavior Section 7.6.4 requires (Appendix D, issue 33).
4. With `changeover`, after the last running output of a group stops, no output of another group may start for `changeover` seconds.
5. An output MUST NOT be in more than one exclusive interlock, nor in more than one group of one.

For example, a system with two heaters and a chiller puts the heaters in one group and the chiller in another, so that the heaters can run together, and neither runs while the chiller does.

13.5 Heaters and Chillers

A heater and a chiller serving one system MUST be kept from running together.

1. When a configuration has an output whose equipment category is `heater` and an output whose equipment category is `chiller`, serving the same system, the two MUST be in different groups of one `exclusive` interlock. The interlock keeps an override, a mode, or a program from starting one while the other runs.
2. Controlling them with one `setpoint` or `proportional` function, as its `raise` and `lower` outputs (Section 11.5), is also RECOMMENDED, because the function keeps a dead band between them.
3. Item 1 does not apply to a heater and a chiller that both regulate themselves (the `thermostat` feature) and are both given their setpoints by one `device-setpoint` function, which keeps the setpoints a dead band apart (Section 11.7). Both are enabled all the time, so an exclusive interlock would keep one of them disabled.

13.6 Leaks and Low Water

Leak and low-water protection, including which equipment stops on a leak, how top-off and water changes respond, and switching heaters off when the water is too low to cover them, is defined by the Water Level and Water Change specification [OAP-WL]. That specification uses the elements of this one: `block` interlocks on level-switch and leak-sensor inputs, latching where a person must check before equipment restarts, and the emergency mode (Section 14.7).

13.7 Safety Functions

A safety function stops equipment that would cause harm if it kept running, and needs at least one path that uses no radio link [OAP-FS Section 13.4]. In a configuration, these elements carry safety functions:

1. **Heater and chiller cutoff:** each function that drives an output whose equipment category is `heater` or `chiller`, except a `device-setpoint` function, whose equipment regulates itself (Section 11.7);

2. **Top-off shutoff:** the top-off functions of the Water Level and Water Change specification [OAP-WL Section 8];
3. **Leak response:** for each system, the interlocks and `emergency` modes whose conditions depend on an input that reads a `leak-sensor` endpoint of that system, which together are one safety function [OAP-WL Section 11.1]; and
4. **Interlocks that stop stop-on-loss equipment:** each interlock that acts on an output whose equipment is stop-on-loss equipment.

A controller:

5. MUST check each of these safety functions for a path that uses no radio link, for each output it stops, as Fail-Safe Behavior Section 13.4 defines, after it starts and whenever its configuration, a binding, or an endpoint's fail-safe configuration changes;
6. MUST, while a safety function has no such path for an output, show its owners the standing warning of Fail-Safe Behavior Section 13.4.6, and raise `safety-path-missing` for each of its elements that stops that output (Section 17.9); and
7. MUST NOT refuse a configuration, or stop running it, because a safety function has no such path.

As Fail-Safe Behavior Section 13.4.3 defines it, an output on a radio link has a path only while it stops by itself on loss of contact, with a `failSafeTimeout` of at most 300 s (provisional), and a safety function with several inputs has one when the inputs whose devices are connected by wire are enough for it to stop the output, whatever its other inputs report, or when it stops the output whenever one of those other inputs is unavailable.

A leak response has a path for an output when one of its elements stops the output with one. For example:

- A heater whose `setpoint` function reads two temperature sensors on Thread, and whose outlet is reached over Wi-Fi and configured to `continue`, has no path: the controller accepts the configuration, and warns until the outlet's fail-safe configuration is changed to `stop`.
- The same heater on an outlet whose `failSafeState` is off and whose `failSafeTimeout` is 3600 s has no path either, and the controller warns until the timeout is 300 s or less.
- A leak emergency mode that starts on `any` of a wired leak sensor and one on Thread has a path for each output on a wired module that it stops, because the wired sensor alone starts the mode, whatever the other reports. With only the sensor on Thread, it would have none, because a lost link hides a leak rather than stopping anything.

14 Modes

14.1 Mode Kinds

A mode is a state of the whole system that changes what several outputs do. Each mode has a `kind`.

Table 21 — Mode kinds

Kind	Precedence	duration	Starts	Ends
<code>emergency</code>	Highest of the modes	Not allowed	By the owner, or while <code>start.when</code> is true	Only by the owner
<code>power-loss</code>	Second	Not allowed	When mains power is lost	When mains power returns
<code>water-change</code>	Third	REQUIRED: the longest the water change may take	By the owner, by <code>start.when</code> , or by the Water Level and Water Change specification	At the end of <code>duration</code> , by the owner, by <code>end.when</code> , or by the Water Level and Water Change specification
<code>maintenance</code>	Fourth	REQUIRED	By the owner, or by <code>start.when</code>	At the end of <code>duration</code> , by the owner, or by <code>end.when</code>
<code>feed</code>	Fifth	REQUIRED	By the owner, or by <code>start.when</code>	At the end of <code>duration</code> , by the owner, or by <code>end.when</code>
An extension name	Below <code>feed</code> , in document order	REQUIRED unless the mode has <code>end.when</code>	By the owner, or by <code>start.when</code>	At the end of <code>duration</code> , by the owner, or by <code>end.when</code>

Every mode ends. A mode of an extension kind **MUST** have a `duration`, an `end.when`, or both; a mode of another kind ends as Table 21 says. A configuration **MAY** have several modes of one kind, such as several feed modes with different durations and actions, and at most one `power-loss` mode. Several `emergency` modes **MAY** be active at once.

14.2 Mode Members

Table 22 — Mode members

Member	Type	Presence	Description
<code>kind</code>	string	REQUIRED	Table 21.
<code>duration</code>	number (s)	see Table 21	How long the mode lasts once started.
<code>start</code>	object	OPTIONAL	<code>when</code> : a condition that starts the mode (Section 14.3).
<code>end</code>	object	OPTIONAL	<code>when</code> : a condition that ends the mode. Not allowed for <code>emergency</code> and <code>power-loss</code> .
<code>outputs</code>	object	OPTIONAL	Output actions, keyed by output identifier (Section 14.4).
<code>commands</code>	array	OPTIONAL	Commands sent once, after the mode starts (Section 14.5). Not allowed for <code>power-loss</code> .

14.3 Starting and Ending Modes

1. A mode with `start.when` starts when that condition changes from false to true. A change from unknown to true does not start it, so a mode does not start late when the clock becomes valid. An `emergency` mode also starts whenever `start.when` is true and the mode is not active, so clearing it while its cause remains starts it again.
2. An unknown condition does not start or end a mode.
3. At most one mode of each kind other than `emergency` is active at a time, except that water-change modes that water change functions of different systems name **MAY** be active together [OAP-WL Section 9.4]. Starting a mode ends any active mode of the same kind that may not be active with it, without its extended states (Section 14.4). If several modes of one kind would start in the same evaluation, only the first in document order starts.
4. The owner **MAY** start a mode again while it is active; its end moves to `duration` from then.
5. A mode with `end.when` ends when that condition changes from false to true.
6. A mode with a `duration` ends at the end of it, unless something ends it earlier.
7. A controller **MUST** record who started each active mode, and keep it with the saved state (Section 17.8): the user's `id` [OAP-SA Section 11.1] for a mode that a user started, such as an owner in an app, a person at the controller, or a Matter bridge's user [OAP-MM Section 12.4.4], and none for a mode that its `start.when`, mains power, or another specification started. It shows it with the mode (Section 17.10). So a Matter bridge can tell a mode that a person started from one that it started itself, and protect the person's mode from home apps.

14.4 Output Actions

`outputs` maps output identifiers to actions. Each action has a `state`, as in Table 4, and an optional `extend`, in seconds.

1. While the mode is active, each output it lists is set to its `state`, at the mode's precedence.
2. Outputs the mode does not list are not affected by it.
3. When the mode ends, each output with `extend` keeps the mode's state for `extend` seconds more, at the same precedence: its extended state. For example, a feed mode can restart the return pump at once and the skimmer ten minutes later.
4. A mode **MUST NOT** list an output that an alarm function drives [OAP-AN Section 15.1]. Modes leave alarm outputs to their alarm function, as power-loss mode does (Section 14.6.2), so that a siren keeps signaling critical alarms, among them the emergency mode's own.

14.5 Mode Commands

`commands` lists commands, each with an `output`, a `command`, its `parameters`, and a `delay`.

1. Each command is sent once, `delay` seconds after the mode starts, if the mode is still active and the output's resolved state is not stop.
2. The command MUST be defined by the output's class or by a feature the output declares. It MUST NOT be `set`, whose states belong in `outputs`, or `dose`, which the Dosing specification governs. It MUST NOT be a command that only a person starts: a dosing head's `prime`, `calibrate`, or `completeCalibration` [OAP-DC Section 7.14.5; OAP-DO Section 10.6.5], or a top-off unit's `resetFault`, which clears faults that only a person clears [OAP-WL Section 7.6].
3. Commands are not sent again after a restart.

A feed mode can, for example, send `feed` to a feeder 30 s after the pumps stop.

14.6 Power-Loss Mode

A `power-loss` mode is the power-loss mode of Fail-Safe Behavior Section 9.5.

1. It is active exactly while the controller detects that mains power is lost: through its own power-fail detection, or, where the mode has `start.when`, while that condition is true.
2. Its `outputs` give each output's state on battery. Fail-Safe Behavior Table 3 gives the RECOMMENDED states: equipment marked `run` keeps running, equipment marked `function` is left to its function, and other equipment stops. A power-loss mode leaves an output to its function by not listing it (Section 14.4.2), so that, for example, an alarm output keeps signaling alarms, among them the loss of mains power [OAP-AN Section 15]. A level MAY slow pumps that keep running, to make the battery last longer.
3. Owners cannot start or end it.
4. When it ends, each output it held stopped stays stopped, at the same precedence, for its endpoint's `powerOnDelay`. The controller then starts these outputs at least 1 s apart, in order of their delays and then of the document, as Fail-Safe Behavior Sections 9.2 and 9.5 require.
5. Interlocks still apply in power-loss mode: an interlock that protects a pump from running dry stops it on battery too.

14.7 Emergency Mode

An `emergency` mode handles an event that needs a person, such as a leak.

1. Only the owner ends it, after checking the system.
2. It takes precedence over overrides and over the other modes, so that, for example, an override that runs the return pump cannot keep pumping water onto the floor. Among several active emergency modes, the first in document order decides an output's state.
3. A controller MUST show an active emergency mode prominently, and MUST raise `emergency-mode` for it while it is active (Section 17.9), which the Alarms and Notifications specification makes a critical alarm [OAP-AN Section 8.6].
4. Its stops have no known end, so the controller latches them (Section 17.5): equipment that it stopped, such as a return pump that would pump water onto the floor, stays stopped if the controller fails, until a controller ends the stop.

The Water Level and Water Change specification defines the recommended response to leaks.

15 Overrides

An override is the owner's manual setting of one output. Overrides are part of the saved state (Section 17.8), with these members:

Table 23 — Override members

Member	Type	Presence	Description
<code>output</code>	identifier	REQUIRED	The output overridden.
<code>state</code>	state	REQUIRED	Run, stop, or a level (Table 4).
<code>start</code>	timestamp	REQUIRED	When the override began.
<code>expires</code>	timestamp	REQUIRED	When it ends. Later than <code>start</code> .

Member	Type	Presence	Description
by	UUID	OPTIONAL	The user who set it: the user's <code>id</code> [OAP-SA Section 11.1], a UUID such as <code>3f6c2a1e-8b4d-4f7a-9c2e-5d1b7a9e0c34</code> .
reason	string	OPTIONAL	The owner's note.

1. Every override MUST have an expiry. Controllers SHOULD propose 1 hour, and SHOULD ask the owner to confirm an expiry more than 24 hours away (provisional).
2. An output has at most one override; a new one replaces it. The owner MAY end an override early.
3. An override takes precedence over functions, programs, and the feed, maintenance, and water-change modes, and is overridden by interlocks and by the emergency and power-loss modes (Section 17.3).
4. Controllers MUST keep overrides across restarts (Section 17.8), and MUST show every active override, with the time left, wherever the output is shown.

An owner who wants equipment off for longer, for example while it is away for repair, changes the configuration instead, which the controller records.

16 Step Programs

16.1 Program Members

A step program changes setpoints, and optionally outputs, step by step over hours or weeks. It serves temporary regimes such as quarantine treatments, slowly lowering salinity, acclimation, and seasonal transitions.

Table 24 — Step members

Member	Type	Presence	Description
name	string	OPTIONAL	Name shown to the owner.
duration	number (s)	REQUIRED, except in the last step	How long the step lasts.
ramp	boolean	OPTIONAL	Whether setpoints change gradually through the step, rather than at its start. Default <code>false</code> .
setpoints	object	OPTIONAL	Setpoint values, keyed by function identifier: a number, or <code>{"offset": n}</code> .
outputs	object	OPTIONAL	Output actions during the step, as in Section 14.4, without <code>extend</code> .
notes	string	OPTIONAL	Free text.

A program has an `id`, an optional `name` and `notes`, and `steps`.

1. Every step but the last MUST have a `duration`.
2. The last step MUST NOT have a `duration` or `ramp`. It holds until the owner stops the program.
3. A program sets the setpoints of `setpoint`, `proportional`, and `device-setpoint` functions only.

16.2 Steps and Ramps

A program run has a start time, and every step's start follows from it and the durations of the steps before it.

1. For each function the program sets, the program's value at any time is:
 - the step's value, from the start of a step that sets it without `ramp`;
 - a straight line from the previous value to the step's value, through a step that sets it with `ramp`; and
 - the previous value, through a step that does not set it.
2. The previous value of the first step is the function's base setpoint at the program's start time (Section 12.1).
3. `{"offset": n}` means `n` plus the function's base setpoint at the program's start time.
4. A step's output actions apply only during that step, at the program's precedence (Section 17.3).

For example, a treatment that raises the temperature over two days, holds it for fourteen, and lowers it over two more, then holds, has four steps: a ramp to the raised value, a step that keeps it, a ramp back, and a final step.

16.3 Starting, Running, and Stopping

1. The owner starts a program, now or at a later time, and stops it. The start time is part of the saved state (Section 17.8).
2. From the start until the owner stops it, the program sets every function that any of its steps sets, even in steps that do not mention it.
3. When the owner stops a program, its functions return to their own setpoints at once, and its output actions end. A program that should return gradually ramps back in its steps before its final step.
4. A program that is waiting for its start time sets nothing.

16.4 Resuming After an Outage

1. A program's position is computed from its start time and the current time, never from saved progress. After any outage, a program whose clock is valid continues exactly where it would have been.
2. While the clock is not valid, a running program holds the values and output actions it last applied, and the controller raises `program-held`. The controller MUST save the values it applies, at least when each step begins and once an hour, for this purpose only.
3. When the clock becomes valid, the program continues from its start time, which may change setpoints at once to where they would have been.

16.5 Programs Together

Several programs MAY run at the same time, but a controller MUST NOT start a program that sets a function or an output that another running program sets.

17 Execution

17.1 Evaluation Cycle

1. A controller MUST evaluate its whole configuration at least once a second (provisional). It MAY also evaluate it when an input changes.
2. At the start of each evaluation, the controller takes the current time and the latest value, quality, and age of every input, and uses only these for the whole evaluation.
3. The controller MUST run its evaluations in the process that also supervises its devices (Fail-Safe Behavior Section 7.1.4), so that devices apply their fail-safe behavior if evaluations stop.

17.2 Evaluation Order

1. Each element is evaluated after every element it refers to. Derived inputs follow their inputs, conditions their inputs and conditions, functions their inputs and conditions, outputs the functions, modes, and interlocks that act on them, and modes their `start.when` and `end.when` conditions.
2. Nothing may depend on itself. A configuration in which an element depends, directly or through other elements, on itself is invalid.
3. An output input of an output bound to an endpoint reads what the endpoint reports, which does not depend on the evaluation, so it creates no dependency. An output input of a virtual output, and a `requires` interlock, depend on the output they name.
4. The outputs of an `exclusive` interlock are resolved together, after every other layer has acted on each of them.
5. Elements that do not depend on each other may be evaluated in any order; the result is the same. Where order matters, as when several modes of one kind would start together, document order decides.

17.3 Precedence

For each output, the controller takes the demand of the highest layer in Table 25 that has one. Interlocks, `stop-all`, and the emergency and power-loss modes rank above a person's override, because an owner who is present can end an emergency mode, and a stop that protects the system should not wait for an override to expire. Overrides rank above the ordinary modes, so that an owner can, for example, run a pump during a feed.

Table 25 — Precedence, highest first

Layer	Demand	Notes
1. Interlocks and stop-all	Stop	Section 13, the stop-all operation [OAP-SA Section 11.8], and the checks that hosted functions apply at this layer (Section 11.10.6). Only stops.
2. Emergency mode	Its action for the output	Section 14.7.
3. Power-loss mode	Its action for the output	Section 14.6.
4. Override	The override's state	Section 15.
5. Water-change mode	Its action for the output	
6. Maintenance mode	Its action for the output	
7. Feed mode	Its action for the output	
8. Modes of extension kinds	Their actions	In document order, the first highest.
9. Step programs	The current step's action	Section 16.2.
10. Control function	The function's demand, after its timing	Section 11; for a lighting or flow function, the light's or pump's program (Section 8.3.2).
11. None	No demand	The output is left as it is.

1. A mode has a demand for an output while it is active and lists the output, and during that output's extended state after it ends. stop-all has a demand, stop, for every output while it lasts, except an output that an alarm function drives, which it leaves to that function [OAP-SA Section 11.8.1].
2. After an output stops, by any layer, it MUST NOT start again until its restartDelay has passed, whatever layer demands it. After the controller starts, restartDelay counts from the start.
3. With no demand, the controller sends the output no command, except to resume control of it (Section 17.4.11), and the output follows its own program, if it has one, or keeps its state, except as item 4 requires.
4. When an output has no control function, and the last layer with a demand on it ends, the controller MUST return the output to the state its endpoint reported just before the first of those layers acted on it, unless the device's own revert does so at the same time (Section 17.5). The controller records that state when such a layer first commands the output, and keeps it with the saved state: one of the states of Table 4, or program for an endpoint that was following its local program. It returns an endpoint to its program as Device Classes Section 7.13 says: with runProgram for a light or a pump, and with set and enabled true for a dosing head or an analyzer. For example, a wave pump that runs its own program, slowed to 40 % on battery, returns to its program when mains power returns.

17.4 Commands

The controller turns each output's resolved state into commands, as items 7 to 11 and Table 5 say, and sends them as items 1 to 6 and 12 say.

A controller:

1. MUST, subject to items 2 and 12, send a command when the endpoint does not report the resolved state, when a temporary state's end changes, and when the endpoint is in its fail-safe behavior (controlSource fail-safe), to resume control of it, unless the resolved state is the fail-safe behavior of an endpoint whose failSafeAction is continue;
2. MUST NOT send commands to an endpoint under local control (controlSource local), except the stops that protect the system, which it MUST send: those of the first layer of Table 25, the checks of hosted functions included (Section 11.10.6); those of the emergency and power-loss modes; and the stop that a function driving a heater or a chiller demands from its input, its cutoff (Section 13.7.1) (Fail-Safe Behavior Sections 8.2, 9.5, and 12);
3. MUST otherwise not send commands only to repeat a state, except as supervision messages that a binding defines;
4. SHOULD NOT send the same command to an endpoint more than once a second, but MUST NOT delay a command that stops an output;
5. MUST follow Fail-Safe Behavior Section 7.4 when it resumes control of an endpoint; and

6. MUST raise `command-refused` when an endpoint refuses a command, except as item 12 says, and SHOULD try again no more than every 5 s.

Each resolved state becomes these commands:

7. Run: the class's `set` command with the output's `runState`.
8. Stop: the class's `set` command with the inactive values, or its `stop` command, as Table 5 says; for a dosing head with the `local-program` feature, both. A stop with no known end is latched as Section 17.5 says.
9. Level: for `level-output`, `set` with `level`; for `pump`, `set` with `speed`, and `running true` above 0 and `false` at 0.
10. Fail-safe: with `failSafeAction state`, the command that sets `failSafeState`; with `continue`, no command.
11. No demand: no command, except to an endpoint in its fail-safe behavior, whose control the controller resumes without changing what the endpoint does, as Device Classes Section 7.13.5 says: `runProgram`, with the parameters that keep what it shows, to a light or a pump that follows its program; `set` with its current `enabled` to a dosing head, an analyzer, or a top-off unit; and, to any other endpoint, the command of the state it reports (items 7 to 9).

Item 11 matters for an endpoint with `continue`, such as a light that runs its own program: after contact returns, the controller resumes control of it, so that it leaves its fail-safe behavior, without taking it off its program.

12. While an endpoint waits out its `powerOnDelay` after its device powers up or restarts, which the controller tells from the device's `uptime` or its binding, it refuses every command but stops [OAP-FS Section 9.1]. The controller MUST send it only stops meanwhile, and the commands that do not change its output, and MUST NOT raise `command-refused` for a command that it refuses then. Once the delay has passed, the controller sends what item 1 asks.

So a skimmer whose `powerOnDelay` is 600 s raises no alarm after every power cut, and the low-water interlock that stops a heater reaches it during the delay too.

17.5 Temporary States, `revertAfter`, and Latched Stops

A temporary state is a state with a known end, set by:

- a mode with a `duration`, until its end, and then the output's extended state, until it ends;
- an override, until it expires;
- a step program's output action, until the end of the step;
- `stop-all`, until it expires [OAP-SA Section 11.8]; or
- a `requires` interlock while the output it requires is stopped by a temporary state, until that state ends and the interlock's `delay` has passed (Section 13.3).

1. When the controller commands an output into a temporary state, it MUST include `revertAfter` with the seconds until that state ends, so that the device ends it even if the controller fails [OAP-DC Section 7.4].
2. When a temporary state's end changes, for example because the owner extends a mode, or when one temporary state follows another, the controller MUST send the command again with the new `revertAfter`.
3. When a temporary state ends, the controller commands the state of the next layer (Section 17.3). A command without `revertAfter` that changes the output cancels any pending revert, and one with `revertAfter` keeps the pending revert's target [OAP-DC Sections 7.4.6 and 7.4.7]. A command that starts or ends an operation, such as a doser's `stop`, leaves a pending revert as it is [OAP-DC Section 7.4.7].

A state without a known end is sent without `revertAfter`: a stop of an interlock, other than one of item 5, and a state of an emergency or power-loss mode, of a mode of an extension kind with no `duration`, or of the last step of a program. Some stops of hosted functions are also sent without `revertAfter`, as their specifications require (Section 11.10.8).

4. A controller MUST latch each stop with no known end that an interlock or an emergency mode demands: it sends the stop's `set` with `latch true`, to every endpoint whose class's `set` accepts `latch` [OAP-DC Section 7.4.13], so that the endpoint keeps the stop through loss of contact and power cuts, in place of its fail-safe behavior, until a controller ends it [OAP-FS Section 8.4]. For a dosing head that runs its own program, the latch goes on the hold of its program, `set` with `enabled false`, which goes with its `stop`. When the stop ends, the controller ends the latch with the command of the next layer, as item 3 says.

5. The stop of a `requires` interlock while the output it requires is stopped by a temporary state has a known end, so the controller sends it with `revertAfter`, for the seconds until that state ends plus the interlock's `delay`, and does not latch it.
6. Other states without a known end, those of power-loss modes, of modes of extension kinds that end only by `end.when`, and of the last step of a program, are neither sent with `revertAfter` nor latched: if the controller fails, each endpoint applies its fail-safe behavior.

For example, while a leak sensor is wet, an interlock stops the return pump with a latched stop. If the controller then fails, the pump keeps the stop after its `failSafeTimeout`, though its fail-safe state runs it, and after any power cut, while the alarm output, which runs on loss of contact, calls a person [OAP-FS Section 8.4]. A skimmer that requires the return pump, which a feed mode pauses for 600 s, is stopped with `revertAfter` 660 when its interlock's `delay` is 60 s, so that it restarts after the return pump even on a failed controller's devices.

An endpoint whose class has no `set`, such as a feeder or a filter roller, cannot be latched, and nor can an endpoint of Device Classes revision 0.4 or earlier, which has no `Latch`. Such an endpoint keeps its fail-safe behavior when its controller fails.

A device of Device Classes revision 0.2 or later keeps the target of a pending revert when a command with `revertAfter` follows [OAP-DC Section 7.4.6]. So after item 2, the device's revert returns the output to the state it had before the first temporary state, even if the controller fails meanwhile, unless the endpoint enters its `failSafeState` first, which cancels the revert [OAP-DC Section 7.4.7]. An endpoint of revision 0.1 returns to the state it had before the latest command [OAP-DC Section 7.4.9], which after item 2 is the earlier temporary state; on such an endpoint, a temporary state that follows another ends on a failed controller's device only through the device's fail-safe behavior.

The commands that end an operation have no `revertAfter`, because an operation that a command ends never resumes [OAP-DC Section 7.14]: the `stop` of a doser, a feeder, or a filter roller. So a temporary stop of a doser, feeder, or filter roller ends the dose, feed, or advance in progress for good, and starts no new one until the controller ends the stop. A temporary stop of a dosing head that runs its own program also holds the program, with `set`, `enabled false`, and `revertAfter`, so the hold ends even if the controller fails [OAP-DC Section 7.13]; the hold and the `stop` may go in either order, because the `stop` leaves the hold's revert as it is. That is safe for dosers and feeders, whose recommended fail-safe configuration is `stop` [OAP-FS Table 3], and of little consequence for a filter roller, which only stops advancing.

17.6 Determinism

Given the same configuration, the same saved state, and the same sequence of evaluation times and input snapshots, two conforming controllers MUST reach the same resolved state for every output. The exceptions are an input value within the rounding error of a unit conversion of a threshold, and the times of sun and moon events, which may differ within the tolerances of Section 10.7.

This makes configurations portable: the same configuration behaves the same way on every controller, and a recorded sequence of inputs can test a controller (Appendix D, issue 12).

17.7 Start-Up and Restart

After power-up or a restart, a controller follows Fail-Safe Behavior Section 9.4, and also:

1. loads its saved state (Section 17.8);
2. if its clock is valid, ends the modes, extended states, overrides, and `stop-all` whose end has passed, and continues the rest;
3. if its clock is not valid, continues each timed mode, extended state, override, and `stop-all` for the time it had left when the state was last saved, measured from the restart, and holds running programs (Section 16.4), because it cannot tell how long it was off;
4. keeps `emergency` modes, latches, latching interlocks, outputs held by `maxOnTime`, and the states to restore (Section 17.3), whatever the clock;
5. takes the first value of each condition as its previous value, so that no mode starts and no `pulse` begins at start-up, except an `emergency` mode whose condition is true; and
6. does not send mode commands again.

17.8 Saved State

The saved state is what a controller MUST keep across restarts. A document's `state` member holds it, so that it can be exported and moved with the configuration. A controller MUST save its state whenever it changes, and SHOULD update `saved` at least every 10 minutes while a temporary state is active, so that after a restart with the clock not valid it knows how much time was left (Section 17.7).

Table 26 — Saved state

Member	Contents
<code>saved</code>	When the state was saved.
<code>modes</code>	Each active mode, and each ended mode whose extended states still last: <code>mode</code> , <code>start</code> , <code>end</code> for a mode with a <code>duration</code> , <code>ended</code> once it has ended, and <code>by</code> , the <code>id</code> of the user who started it, if a user did (Section 14.3.7).
<code>overrides</code>	Each active override (Section 15).
<code>programs</code>	Each program run: <code>program</code> and <code>start</code> .
<code>latched</code>	Latches that are set, latching interlocks that hold their outputs, and outputs held by their function's <code>maxOnTime</code> .
<code>restore</code>	For each output without a function that a layer has acted on, the state to return it to: a state of Table 4, or <code>program</code> (Section 17.3.4).
<code>stopAll</code>	While <code>stop-all</code> lasts: its <code>start</code> , its <code>expires</code> , and <code>by</code> , the <code>id</code> of the user who started it [OAP-SA Section 11.8].
<code>functions</code>	The saved state of each hosted function that keeps one, keyed by the function's <code>id</code> , in the form that its specification defines (Section 11.10).

Hysteresis decisions, timers, and timing limits are not saved; they start afresh as Sections 10 and 11 describe. The stops of interlocks are not saved either: the controller computes them again from their conditions, and the endpoints keep their latched stops meanwhile (Section 17.5). A restart that forgot `stop-all` would command the return pump and the heaters back on while a person still wants them stopped, so `stop-all` is saved.

17.9 Faults

A controller raises the faults in Table 27, and clears each when its condition ends, unless the table says the owner clears it. Each fault names the element it concerns. How faults become alarms and notifications is defined by the Alarms and Notifications specification.

Table 27 — Automation faults

Code	Meaning
<code>input-unavailable</code>	An input used by a function, interlock, or mode is unavailable (Section 9.8).
<code>inputs-disagree</code>	The inputs of a derived input differ by more than its <code>maxDifference</code> (Section 9.7).
<code>max-on-time</code>	A function stopped an output that reached its <code>maxOnTime</code> . Without <code>restTime</code> , the owner clears it (Section 11.8).
<code>interlock-latched</code>	A latching interlock holds outputs stopped until the owner clears it (Section 13.2).
<code>binding-missing</code>	An output or input has no endpoint, or its endpoint does not match it (Section 8.5).
<code>command-refused</code>	An endpoint refused a command (Section 17.4).
<code>program-held</code>	A step program cannot advance because the clock is not valid (Section 16.4).
<code>unsupported</code>	The controller does not support an element of the configuration (Section 19.2).
<code>emergency-mode</code>	An emergency mode is active (Section 14.7).
<code>safety-path-missing</code>	A safety function has no path that uses no radio link to an output it stops (Section 13.7).

17.10 Visibility

A controller MUST show the owner, for every output:

1. its resolved state, and which layer of Table 25 decided it, by name, such as "stopped by interlock 'Sump low water'", "stopped by stop-all, 52 min left", or "feed mode 'Feed A', 4 min left";
2. every active override and temporary state, with the time left; and
3. for a function with a setpoint, the effective setpoint and where it comes from.

It MUST also show active modes and programs, with the time left and the current step, who started each mode (Section 14.3.7), and every fault of Table 27.

Owners need to know why equipment is doing what it does, especially when two rules disagree.

18 Presets

18.1 Using Presets

A preset is a recommended set of elements for one kind of equipment, by Data Model equipment category: a control function, interlocks, and actions in each mode. `presets.json` contains them in machine-readable form, and Tables 28 and 29 are generated from it.

1. When the owner assigns equipment to an output, a controller SHOULD offer the preset for its category, and create elements as the preset describes them, after the owner confirms them.
2. A preset never sets a target value for the water, such as a temperature or pH setpoint, or a limit. The owner sets the values listed under “Owner sets”, and a controller MUST NOT run a function created from a preset until the owner has set them. Targets are the owner’s own, entered or confirmed by the owner, as ranges are [OAP-AN Section 9.1.4].
3. The values a preset does give, such as hysteresis and delays, protect equipment and are provisional.
4. Each output’s fail-safe configuration follows Fail-Safe Behavior Table 3. The “Power loss” column of Table 29 repeats that table’s “On battery” column, from `recommendations.json`, in which `function` means that the power-loss mode leaves the output to its function (Section 14.6.2).
5. A function created from a preset records it in `preset`, so that another controller can recognize it.

Presets for lighting, flow, dosing, top-off, water changes, RO/DI, and analyzers give only their mode actions; their control is the hosted function type that each names, which its own specification defines (Section 11.10). The fan preset uses the equipment category `fan` [OAP-DM], for which Fail-Safe Behavior Table 3 gives a recommendation. Alarm outputs have no preset: an alarm function drives them [OAP-AN Section 15].

18.2 Preset Tables

In Table 28, “Owner sets” lists what the owner must give before the function runs. In Table 29, `follow` means that the mode leaves the output alone, and the interlocks are those the controller should offer.

Table 28 — Presets: control

Preset	Category	Output classes	Control	Values	Owner sets	Unavailable input
heater	heater	heater, switch	setpoint, raising, on temperature; the higher of two sensors	hysteresis 0.5 degC; <code>maxDifference</code> 1.0 degC	setpoint, limits	fail-safe
chiller	chiller	chiller, switch	setpoint, lowering, on temperature; the higher of two sensors	hysteresis 0.5 degC; <code>deadBand</code> 1.0 degC; <code>restartDelay</code> 300 s	setpoint, limits	fail-safe
return-pump	return-pump	pump, switch	fixed, run; or a <code>flow</code> function, for a return pump that holds a constant flow (Flow Programs specification)	—	—	—
protein-skimmer	protein-skimmer	switch, pump	fixed, run	—	—	—
calcium-reactor	reactor	valve, switch	setpoint, lowering, on pH; the CO2 supply, on the pH of the reactor’s effluent	hysteresis 0.05 pH	setpoint, limits	fail-safe
kalkwasser-reactor	reactor	switch, pump, valve	condition, on a schedule	—	schedule	stop

Preset	Category	Output classes	Control	Values	Owner sets	Unavailable input
ozone-generator	ozone-generator	switch	setpoint, raising, on orp	hysteresis 10 mV	setpoint, limits	fail-safe
co2-system	co2-system	valve, switch	setpoint, lowering, on ph	hysteresis 0.05 pH	setpoint, limits	fail-safe
uv-sterilizer	uv-sterilizer	switch	fixed, run	—	—	—
fan	fan	level-output, switch	proportional, lowering, on temperature; or a setpoint function, lowering, with hysteresis 0.5 degC, for a fan on a switch	band 1.0 degC; minLevel 30 %	setpoint	fail-safe
air-pump	air-pump	switch, level-output	fixed, run	—	—	—
light	light	light, level-output, switch	A lighting function, defined by the Lighting Programs specification	—	—	—
circulation-pump	circulation-pump	pump, level-output, switch	A flow function, defined by the Flow Programs specification	—	—	—
doser	doser	doser	A dosing function, defined by the Dosing specification	—	—	—
auto-top-off	auto-top-off	auto-top-off, switch, valve, pump	A top-off function, defined by the Water Level and Water Change specification	—	—	—
water-changer	water-changer	pump, valve, doser, switch	A water-change function, defined by the Water Level and Water Change specification	—	—	—
ro-di	ro-di	valve, switch	A ro-di function, defined by the Water Level and Water Change specification	—	—	—
feeder	feeder	feeder	A feed command from a feed mode	—	—	—
analyzer	analyzer	analyzer	A water-testing function, defined by the Water Testing specification	—	—	—

Table 29 — Presets: modes and interlocks

Preset	Feed	Maintenance	Water change	Emergency	Power loss	Interlocks	Notes
heater	follow	stop	stop	stop	stop	stops on low water (Water Level and Water	Use two temperature sensors, acting on

Preset	Feed	Maintenance	Water change	Emergency	Power loss	Interlocks	Notes
						Change); never with the <code>chiller</code>	the higher. Set the heater's own thermostat about 1 °C above the setpoint as a backstop. With a chiller, use one function for both, with a dead band.
<code>chiller</code>	<code>follow</code>	<code>stop</code>	<code>stop</code>	<code>stop</code>	<code>stop</code>	never with the <code>heater</code> ; requires the <code>return-pump</code> , or the pump that feeds the chiller, running for 30 s	The restart delay protects the compressor; follow the maker's instructions where they ask for longer. The dead band applies when one function controls both the heater and the chiller.
<code>return-pump</code>	<code>stop</code>	<code>stop</code>	<code>stop</code>	<code>stop</code>	<code>run</code>	stops on low water in the sump (Water Level and Water Change)	Feed modes send the pause with <code>revertAfter</code> , so the pump restarts even if the controller fails. A stop for a leak is latched, so the pump stays stopped even if the controller fails.
<code>protein-skimmer</code>	<code>stop, then 600 s more</code>	<code>stop</code>	<code>stop</code>	<code>stop</code>	<code>stop</code>	requires the <code>return-pump</code> , running for 60 s; stops while the collection cup is full	Starting after the return pump, and some minutes after feeding, lets the sump level settle and keeps the skimmer from overflowing.
<code>calcium-reactor</code>	<code>follow</code>	<code>stop</code>	<code>follow</code>	<code>stop</code>	<code>stop</code>	requires the reactor's recirculation pump; stops while <code>ph</code> in the display is below the owner's limit	The function controls the CO2 supply. The recirculation pump is a separate output, usually with a <code>fixed</code> function.
<code>kalkwasser-reactor</code>	<code>follow</code>	<code>stop</code>	<code>stop</code>	<code>stop</code>	<code>stop</code>	stops while <code>ph</code> in the display is above the owner's limit; requires the <code>return-pump</code>	The block uses <code>onUnavailable</code> block, so the reactor stops when the pH reading is lost. Amounts delivered by a doser are set as defined by the Dosing specification.
<code>ozone-generator</code>	<code>follow</code>	<code>stop</code>	<code>stop</code>	<code>stop</code>	<code>stop</code>	requires the <code>protein-skimmer</code> , running for 300 s; stops while <code>orp</code> is	Ozone is usually fed through the skimmer, so it stops whenever

Preset	Feed	Maintenance	Water change	Emergency	Power loss	Interlocks	Notes
						above the owner's limit	the skimmer stops.
co2-system	follow	stop	follow	stop	stop	stops while ph is below the owner's limit	In planted systems, CO2 usually also stops outside the photoperiod: add a schedule to the block, or use a condition function.
uv-sterilizer	follow	stop	follow	stop	stop	requires the return-pump, or the pump that feeds the sterilizer	A lamp without flow overheats. Stopping it during maintenance protects the owner's eyes. Medication programs often stop it too.
fan	follow	follow	follow	follow	stop	—	A fan left running cools the water by a few degrees at most, which the heater can make up, so it keeps running if its controller is lost.
air-pump	follow	follow	follow	follow	run	—	Aeration keeps the water oxygenated, and is useful in every mode.
light	follow	follow	follow	follow	stop	—	
circulation-pump	stop	stop	stop	follow	run	—	Flow within the display does not move water out of it, so it continues during an emergency.
doser	follow	stop	stop	stop	stop	—	Whether a dose missed during a mode is given later is defined by the Dosing specification.
auto-top-off	stop, then 300 s more	stop	stop	stop	stop	—	While the return pump is paused, the sump level is not its normal level, so top-off waits until it has settled.
water-changer	stop	stop	follow	stop	stop	—	It carries out the water change itself.
ro-di	follow	stop	follow	stop	stop	—	
feeder	follow	stop	stop	stop	stop	—	A feed mode sends feed once the pumps have stopped, for

Preset	Feed	Maintenance	Water change	Emergency	Power loss	Interlocks	Notes
							example 30 s after it starts.
analyzer	follow	stop	stop, then 1800 s more	follow	stop	—	Samples taken while the water is being changed or worked on are not representative, and new water takes a while to mix (Water Testing specification).

19 Portability

19.1 Export

1. A controller **MUST** be able to export its complete configuration and saved state as a configuration document, at any time, without a network service.
2. The export **MUST** include every element, binding, and saved state the controller holds, and every extension it has preserved.

19.2 Import

1. A controller **MUST** validate a document before using it, and **MUST** reject it, without changing its running configuration, if it is not valid.
2. A controller **MUST** support every element type defined by this Specification.
3. An element of a type the controller does not support, such as an extension type from another maker, is unsupported. The controller **MUST** tell the owner before it accepts the document, **MUST** raise `unsupported`, and **MUST** treat an unsupported input or condition as unavailable and an unsupported function as one whose input is unavailable. An unsupported interlock holds its outputs stopped, as a `block` whose condition is unknown, and an unsupported mode never starts. It **MUST NOT** run a configuration it only partly understands as if it were complete.
4. Bindings to devices the controller does not have, or that fail Section 8.5, are unusable. The controller **MUST** ask the owner to bind each output and endpoint input that has no usable binding, and **SHOULD** propose endpoints whose class, features, parameter, and `loadCategory` match.
5. The owner **MUST** confirm an imported configuration before the controller runs it (Section 21).
6. Saved state is imported with the configuration, so that modes, overrides, and programs continue. For example, a quarantine treatment moved to a new controller continues from its start time.

19.3 Sharing

A document without `bindings` and `state` describes automation without any particular hardware, and can be shared with other owners, for example to pass on a treatment program or a whole configuration. Before sharing, a controller **SHOULD** offer to remove the location, names, and notes (Section 21).

20 Conformance

20.1 Configuration Documents

A conforming configuration document:

1. is valid against the JSON Schema (Section 7.1); and
2. meets every requirement of Sections 7 to 18 that applies to documents, including those a schema cannot express, such as references, units, the evaluation order, setpoint limits, and the rule for heaters and chillers.

20.2 Controllers

A conforming controller:

1. imports, exports, and validates configuration documents (Section 19);
2. evaluates configurations as Sections 8 to 17 define, including precedence, unavailable inputs, temporary states, latched stops, and restarts;
3. keeps its saved state across restarts (Section 17.8);
4. raises the faults of Table 27, and shows what Section 17.10 requires;
5. warns its owners as Sections 9.8 and 13.7 require;
6. implements the Fail-Safe Behavior specification for controllers [OAP-FS Section 14.2];
7. requires the authorization of Section 21, item 3; and
8. passes the test procedures of Section 20.4.

There is one conformance level for controllers: a controller conforms when it implements every requirement of this Specification that applies to it, with no reduced profile for small controllers, as lights and pumps that run their own programs must implement every program their schemas allow.

20.3 Schema and Tools

The JSON Schema, `schema/automation.schema.json`, is normative for document structure. It refers to the Data Model schema for systems, equipment, and timestamps. Where the schema and this document differ, this document governs.

`presets.json` is normative for the presets, and Tables 28 and 29 are generated from it by `tools/render_spec_tables.py`.

The validator, `tools/validate_automation.py`, checks documents against the schema and the rules a schema cannot express. It is informative: a document is conforming if it meets this Specification, whether or not the tool accepts it.

The `examples` directory contains complete configurations (Appendix B).

20.4 Test Procedures

Tests use a controller with modules or simulated devices whose inputs the tester can set, and whose clock the tester can set or stop. Each test names the requirements it checks. Every MUST and MUST NOT of Sections 7 to 19 and 21 is checked by at least one test. A pass criterion that checks what a SHOULD asks applies to a product that does it.

Table 30 — Test procedures

Test	Requirements	Method	Pass criteria
Import and export	7.4, 7.8, 19.1, 19.2.1, 19.2.2, 19.2.5, 19.2.6, 21.2	With the controller's internet access blocked, import each example in <code>examples</code> , and one of them with <code>oapAutomation "0.2"</code> , bind it, and export it. Then import a copy of <code>minimal.json</code> with <code>oapAutomation "0.1"</code> , a reservoir without <code>circulating</code> , and an override whose <code>by</code> is <code>alex</code> , and export it.	Each is accepted only after the controller has shown which outputs it drives and how, and the owner has confirmed it and every binding the controller did not make. The export is valid, has <code>oapAutomation "0.3"</code> , and is equal to the import apart from <code>created</code> , <code>generator</code> , <code>state.saved</code> , and bindings the owner made. The copy of revision 0.1 is read with the reservoir circulating, counted in the system's water volume, and the override without <code>by</code> .
Invalid documents	7.1 to 7.7, 8.1, 8.5.5, 9.2, 9.5.1, 10.7.2, 10.7.3, 11.1, 11.4, 11.6, 11.7.1, 11.7.3, 11.9, 11.10.4, 12.2.5, 12.3, 12.4.1 to 12.4.3, 13.3.3, 13.4.5, 13.5, 14.1, 14.4.4,	Import documents that each break one rule of Sections 7 to 18 for documents, such as: a reference to a missing input; an input whose unit is not accepted for its parameter; an element that depends on itself; an <code>id</code> used twice; a <code>posixTimeZone</code> that differs from <code>timeZone</code> ; a setpoint value outside its <code>limits</code> ; a heater and a chiller of	Each invalid document is rejected, and the running configuration is unchanged. The markup is shown as text.

Test	Requirements	Method	Pass criteria
	14.5.2, 16.1, 17.2.2, 19.2.1, 21.1	one system in one group, or in no exclusive interlock; a mode of an extension kind with neither <code>duration</code> nor <code>end.when</code> ; a mode command <code>set</code> , and one <code>prime</code> ; a mode that lists the output of an alarm function; <code>noonAt</code> with <code>shift</code> ; a <code>location</code> at latitude 65°; a dosing function that its own schema rejects; and a last program step with a <code>duration</code> . Also import a valid document whose names hold markup.	
Unsupported elements	7.7.3, 19.2.3	Import a configuration with an input, a function, an interlock, and a mode of extension types or kinds that the controller does not support.	The owner is told before accepting it; <code>unsupported</code> is raised; the function's outputs take its unavailable-input behavior; the interlock holds its outputs stopped; the mode never starts.
Converting units	7.5.2	With a product that converts configurations, convert one with a setpoint of 25.0 degC and a hysteresis of 0.5 degC to degF.	The setpoint is 77.0 degF and the hysteresis 0.9 degF.
Bindings	8.5.6, 8.5.7, 19.2.4	Bind an output to an endpoint that lacks a feature the output declares, and an input to a sensor of another parameter; then import a configuration whose bindings name devices the controller does not have.	Neither binding is used; <code>binding-missing</code> is raised for each output and input without a usable binding; the controller asks the owner to bind each of them.
Setpoint control	11.4	Raise and lower the input of a <code>setpoint</code> function slowly across its setpoint.	The output switches at the values of Table 15, within the input's resolution.
Heating and cooling	11.5, 13.5	With a heater and a chiller in one function and an exclusive interlock, sweep the temperature across the dead band while the setpoint changes, and override each output in turn.	The heater and the chiller never run together. They switch at the values of Table 15.
Exclusive	13.4	Make two outputs of an exclusive interlock demand to run at once, then in turn. Then give an interlock two heaters in one group and a chiller in another, and make all three demand to run.	Only one runs at a time; the one already running keeps running; <code>changeover</code> is respected. The two heaters run together, and the chiller runs only once neither heater does.
Lost devices	9.3.3, 13.3.1, 13.4.3	Put a heater and a chiller in different groups of an exclusive interlock, the heater on a network device whose outlet has <code>failSafeAction</code> state and a <code>failSafeTimeout</code> of 120 s, and a skimmer that requires the heater's outlet. Disconnect the device while the heater runs and the chiller is called for. Repeat with the outlet on <code>failSafeAction</code> <code>continue</code> , disconnecting it 60 s after a maintenance mode stopped the heater with <code>revertAfter</code> 600.	Until 122 s after the controller's last valid message to the device, an <code>output</code> input of the heater is unavailable, the chiller does not start, and the skimmer is stopped; after that, the chiller starts. With <code>continue</code> , the heater counts as stopped from 122 s, so the chiller starts, until its revert would have fallen due, when it counts as running, and the chiller is held stopped again.
Redundant sensors	9.7.3, 9.7.4	With a <code>maximum</code> of two temperature inputs, raise one sensor above the other, then separate them by more than <code>maxDifference</code> , then disconnect one, then both.	The function acts on the higher; <code>inputs-disagree</code> is raised; it continues on the remaining sensor with <code>input-unavailable</code> ; then it applies its unavailable-input behavior.
Qualified readings	9.2.2	Make a conductivity sensor report > 80 mS/cm with <code>quality</code> good, read by a <code>compare</code> condition with above 70 and by a <code>setpoint</code> function; then report it with <code>quality</code> out-of-range.	The comparison is true, and the setpoint function treats the input as unavailable; with <code>out-of-range</code> , the comparison is unknown.

Test	Requirements	Method	Pass criteria
Analyzer and observation inputs	9.2.3, 9.5.4, 9.5.5	Bind an endpoint input to an analyzer that a testing function drives, and use an <code>observation</code> input of nitrate. Make the analyzer report a result that the testing function rejects, then one that it accepts and corrects. Then record a nitrate result whose <code>unitAssumed</code> is <code>true</code> , and one dated a day after the controller's clock.	The <code>endpoint</code> input reads only the accepted result, corrected, with its sample time. The <code>observation</code> input passes over the rejected result, the result with an assumed basis, and the result dated ahead, and reads the latest of the others, until a person confirms the basis, or the clock reaches the later date.
Stale readings	9.8.7	Make a sensor keep reporting the same value with <code>quality</code> <code>good</code> and an <code>updated</code> that does not change.	The input becomes unavailable once the reading is older than <code>maxAge</code> .
Default maximum ages	9.8	Bind a sensor input without <code>maxAge</code> to a sensor that reports a <code>measurementInterval</code> of 300 s, on a Device Protocol device that returned a <code>maxInterval</code> of 600 s, and stop its readings; then configure a <code>maxAge</code> of 120 s.	The input stays available until its reading is 1800 s old. The controller warns the owner about the <code>maxAge</code> of 120 s.
Calibration hold	9.8.6	Start a calibration of a probe whose input a function uses, finish it, and confirm the probe. Then unplug the probe and plug it back in. Then make another endpoint of its port active, and back.	The input is unavailable, and the function applies its unavailable-input behavior, until each hold ends: after the calibration, until the probe is confirmed and has settled; after the probe is reconnected, and after the active endpoint changes, until its readings have settled.
Unavailable-input behaviors	11.9	For each value of <code>onUnavailable</code> in Table 18, make the function's input unavailable.	The output does as Table 18 says, at once, and <code>input-unavailable</code> is raised.
Logic	10.3	Set the inputs of <code>all</code> , <code>any</code> , and <code>not</code> conditions to every combination in Table 12.	The results match Table 12.
Latches	10.4	Set a latch that has no <code>reset</code> , restart the controller, then try to clear it with <code>set</code> still true, then with it false.	It stays set across the restart and while <code>set</code> is true; the owner can clear it when <code>set</code> is false.
Timers	10.5	Drive the input of each timer kind with pulses shorter and longer than <code>duration</code> .	Each follows Section 10.5, within 1 s.
Schedules	10.6	Run windows across midnight, with <code>days</code> , <code>everyDays</code> , and <code>dates</code> , with the clock set to the days around each edge.	Each window starts and ends on the right days, within 1 s.
Daylight saving	7.2, 7.6.7, 10.6.6	On a controller that carries the Time Zone Database, in a time zone with daylight saving time, run windows that start and end in the hour skipped and the hour repeated.	They follow Section 7.6, in the time zone of <code>timeZone</code> .
Clock not valid	12.5, 16.4.2, 17.7.3	Restart the controller with its clock not valid.	Schedules are unknown; setpoints use their <code>fallback</code> or the unavailable-input behavior; programs hold the values they last applied, with <code>program-held</code> ; timed modes and overrides last for the time they had left when the state was saved.
Sun and moon	10.7.3, 10.7.4, 10.7.5	Compute events and moon illumination for three locations, at latitudes near 0°, 35°, and 55°, on four dates, with no shift, with <code>noonAt</code> 13:00, and with <code>shiftDays</code> 182.	Sun events are within 2 s, moon events within 60 s, and moon illumination within 0.1 percentage points, of the method <code>oap-1</code> [OAP-LP Section 9.6], with the shift of Lighting Programs Section 9.3 on each date.
Proportional control	11.6	Sweep the input of a <code>proportional</code> function through its band and back.	The level follows Section 11.6 within 1 percentage point, and stops as <code>hysteresis</code> says.

Test	Requirements	Method	Pass criteria
Device setpoints	11.7.4	Change the effective setpoint of a <code>device-setpoint</code> function by 0.05 °C, then by a further 0.1 °C.	The first change is not written; the second is written to the endpoint's <code>setpoint</code> , in its unit, within a minute.
Temporary setpoints	11.7.6, 11.7.7	Give a <code>device-setpoint</code> function of a heater of Device Classes revision 0.5 an offset of +1.0 °C for two days with a <code>ramp</code> of 3600 s; then stop the controller's evaluations during the ramp. Then, with the controller running, run a step program of two steps, the second its last.	During the offset and the first step, each setpoint is sent with <code>set</code> , <code>setpoint</code> , and <code>revertAfter</code> for the time until the offset or the step ends. With the evaluations stopped, the heater returns to the setpoint from before the offset when the revert falls due. The last step's setpoint is sent with <code>set</code> and no <code>revertAfter</code> , and later changes are written to the attribute.
Setpoint limits and offsets	12.1, 12.3, 12.4.4	Give a function a <code>daily</code> table and an offset with a <code>ramp</code> that takes the effective setpoint past <code>limits.max</code> , then let the offset end.	The effective setpoint follows the table, grows with the ramp, is held at <code>limits.max</code> , which the controller shows, and returns to the table when the offset ends.
Timing	11.8	Drive a <code>condition</code> function with every timing member, including <code>maxOnTime</code> with and without <code>restTime</code> , with a restart while it is held.	Each follows Section 11.8; a function held without <code>restTime</code> stays held across the restart until the owner clears <code>max-on-time</code> .
Restart delay	17.3.2	Stop an output with <code>restartDelay</code> by an override, then end the override.	The output does not start until <code>restartDelay</code> after it stopped.
Requires	13.3	Stop the required output, then start it with its endpoint refusing the command, then let it start.	The dependent output stops in the same evaluation; it does not start while the required one is not reported running; it starts <code>delay</code> after it is.
Precedence	13.1, 17.3	For each pair of layers in Table 25, make both demand different states of one output.	The higher layer decides, and the controller shows which layer it is.
Stop-all	17.3.1, 17.4.2, 17.5, 17.7, 17.8	Set an override that runs a pump, start a local control on another device, and drive a bound siren with an alarm function; then use <code>stop-all</code> with an expiry of 10 minutes; restart the controller after 5 minutes, and let <code>stop-all</code> expire.	Every output but the siren stops, the one under local control included; the siren keeps signaling the critical <code>stop-all</code> alarm. The stops carry <code>revertAfter</code> , except the commands that end an operation. <code>stop-all</code> survives the restart, and the controller shows it with the time left; when it expires, 10 minutes after it began, each output returns to the state the other layers give it.
Temporary states	17.1.3, 17.5.1, 17.5.2	Start a feed mode that pauses one pump whose <code>failSafeAction</code> is <code>continue</code> and one whose <code>failSafeAction</code> is <code>state</code> , extend it, then stop the controller's evaluations while its devices remain powered.	The extension is sent again with the new <code>revertAfter</code> , and devices stop receiving supervision messages. The pump with <code>continue</code> ends the mode's state when the mode would have ended, by <code>revertAfter</code> , returning to the state from before the mode. The pump with <code>state</code> applies its fail-safe behavior at its timeout, which cancels the revert if it comes first.
Latched stops	14.7.4, 17.5.4, 17.5.5, 17.5.6	With a return pump of Device Classes revision 0.5 whose fail-safe state runs it, and a skimmer that requires it with <code>delay</code> 60, make a leak sensor wet that an interlock on the pump and an emergency mode read; then stop the controller; then cut and restore the pump's power. With the controller running again and the leak gone,	The pump's stops are sent with <code>latch true</code> , and it stays stopped after its fail-safe timeout and after the power cut. In the feed mode, the skimmer's stop is sent with <code>revertAfter</code> 660 and without <code>latch</code> . The power-loss mode's stop has neither <code>revertAfter</code> nor <code>latch</code> .

Test	Requirements	Method	Pass criteria
		start a feed mode that pauses the return pump for 600 s, then a power-loss mode that stops it.	
Extended states	14.4.3	End a feed mode in which the skimmer has <code>extend</code> .	The other outputs resume at once; the skimmer resumes <code>extend</code> seconds later.
Mode commands	14.5	Start a feed mode with a <code>feed</code> command delayed 30 s; start it again with the feeder held stopped by an interlock; restart the controller during a third run.	The command is sent once, 30 s after the start, and not when the feeder's resolved state is stop; it is not sent again after the restart.
Restoring outputs	17.3.4	Slow a pump that has no function, and runs its own program, with a power-loss mode; then restore mains power.	The pump returns to its program, by <code>runProgram</code> .
Resuming control	17.4.1, 17.4.11, 17.4.5	Disconnect a light that runs its own program, with <code>failSafeAction continue</code> and no demand on it, for longer than its timeout, then reconnect it.	The controller reads the light's state, then sends <code>runProgram</code> with the parameters that keep what it shows; <code>controlSource</code> becomes <code>controller</code> , and the light stays on its program.
Emergency	14.3, 14.7	Trigger an emergency mode from a leak input; clear it while the leak is still detected, then after it is dry. Then trigger two emergency modes at once.	It starts again while the leak remains, and ends only when the owner clears it after the leak is dry. It is shown prominently, with <code>emergency-mode</code> raised for each mode while it is active. Two emergency modes stay active together.
Repeated commands	17.4.3, 17.4.4	Hold an output's demand steady for 10 minutes, then send it a run and, within a second, a stop from an interlock.	While the endpoint reports the resolved state, the controller sends it nothing but the binding's supervision messages. The stop is sent at once.
Local control	17.4.2	Start a local control on a device, then bring about a function's demand, an override, an interlock, and a power-loss mode that stops the output. Then switch a heater's outlet on at its own controls, and let the water pass the heater function's setpoint.	The controller sends nothing, except the stops of the interlock and the power-loss mode, and the heater function's stop at its setpoint.
Power-on delays	17.4.6, 17.4.12	Cut and restore the power of a module with a skimmer outlet and a heater outlet, each with a <code>powerOnDeLay</code> of 600 s, while a low-water interlock holds the heater stopped.	During the delay, the controller sends the heater its stop, and the skimmer nothing, and raises no <code>command-refused</code> ; once the delay has passed, it starts the skimmer.
Power loss	14.6	Remove and restore mains power, with an interlock that holds stopped an output configured to run on battery, and an alarm output that the power-loss mode does not list.	Outputs take their battery states, except the one the interlock holds stopped; the alarm output follows its alarm function and signals the loss of mains power; when power returns, outputs restart as Fail-Safe Behavior Section 9.2 requires.
Overrides	15, 17.7	Set an override, restart the controller with a valid clock, then let it expire; set one and restart with the clock not valid.	The override has an expiry, and is shown with the time left. It survives the first restart and ends on time; after the second restart it lasts for the time it had left.
Step programs	16.2, 16.3, 16.4	Run a program with a ramp; switch the controller off for 2 hours during the ramp, with its clock kept valid.	After the restart, the setpoint equals the program's value for the current time.
Program conflicts	16.5	Start a program that sets a function already set by a running program.	The controller refuses it.
Saved state	10.4.5, 14.3.7, 17.8	Cut the controller's power just after each kind of change of the saved state: a mode starts, started by a user, an override is set, a	Each is restored after the restart, with the user who started the mode.

Test	Requirements	Method	Pass criteria
		program starts, a latch is set, a latching interlock latches, a state to restore is recorded, <code>stop-all</code> starts, and a hosted function's state changes.	
Start-up	10.2, 17.7	Restart the controller while a scheduled feed mode's start condition is true. Then, while running, make the input of a <code>compare</code> condition that starts a mode unavailable, and available again beyond its threshold.	The feed mode does not start, and no command is sent before the conditions of Fail-Safe Behavior Section 9.4 are met. The mode does not start when the input returns.
Evaluation	17.1.1	Step the input of a <code>compare</code> condition that drives an output across its threshold, many times at random moments.	Each change is commanded within 1 s, plus the binding's own delay.
Determinism	17.6	Run one recorded sequence of evaluation times and input snapshots through two controllers with the same configuration and saved state.	They reach the same resolved state for every output in every evaluation, except within the exceptions of Section 17.6.
Presets	18.1	Create a function from the <code>heater</code> preset, and try to run it before the owner sets its setpoint and limits.	The controller offers the preset's elements for the owner to confirm; the function does not run until the owner has set them; it records <code>preset</code> .
Safety functions	13.7	Bind a heater's <code>setpoint</code> function's output to an outlet reached over Wi-Fi, with <code>failSafeAction continue</code> , no thermostat of its own, and no backup thermostat; restart the controller; then give the outlet <code>failSafeAction</code> state with its inactive state and a <code>failSafeTimeout</code> of 3600 s, then of 120 s. Then start a leak emergency mode on any of a wired leak sensor and one on Thread, stopping a pump on a module; then remove the wired sensor from it.	The configuration is accepted and runs. The controller shows every owner a standing warning, and raises <code>safety-path-missing</code> for the function, from start-up until the outlet's timeout is 120 s. The leak mode raises none while it reads the wired sensor, and raises it once only the sensor on Thread is left.
Faults	17.4.6, 17.9	Bring about each fault of Table 27.	Each is raised, naming its element, and clears when its condition ends, or, where Table 27 says so, when the owner clears it.
Visibility	13.1.4, 14.3.7, 15.4, 17.10	Bring about each layer of Table 25 in turn, with a function that has a setpoint, a running program, and active modes.	The controller shows, for each output, its state and the layer and element that decided it; every override and temporary state with the time left; the effective setpoint and its source; active modes and programs with the time left and the current step, and who started each mode; and every active fault.
Authorization	21.3	As a user with the <code>view</code> role, try to set an override, start a mode, and clear a latch; as a user with the <code>override</code> role, try to change the configuration and start a step program.	Each is refused.
Resource limits	21.7	Import a configuration that exceeds a limit the controller documents, such as its number of elements.	It is rejected, and the controller says which limit it exceeds.

21 Security Considerations

1. **Untrusted documents.** Configuration documents are untrusted input. Controllers MUST validate them fully before use, SHOULD limit their size, and MUST escape strings before displaying them. The format contains no program code, by design, so a document cannot run code on a controller.

2. **Imported automation controls equipment.** A configuration from another owner or another controller can stop a return pump or run a heater. A controller **MUST** show the owner what an imported configuration will do, at least which outputs it drives and how, and **MUST** have the owner confirm it, and every binding it did not make itself, before running it.
3. **Authorization.** Changing a configuration, starting and ending modes and programs, setting overrides, and clearing latches **MUST** require authorization as defined by the Security and Access specification. Starting a mode or setting an override is a smaller permission than changing the configuration. Each action needs one of its operations [OAP-SA Table 6]:
 - `change-configuration`, to change or import a configuration;
 - `start-program`, to start or stop a step program;
 - `set-override`, to set or end an override;
 - `start-mode`, to start a mode of any kind, or to end one other than an emergency mode;
 - `end-emergency-mode`, to end an emergency mode;
 - `clear-latch`, to reset a latch, clear a latching interlock, or clear `max-on-time`; and
 - `stop-all`, to stop every output.
4. **Protection does not depend on automation.** A configuration cannot change a device's fail-safe configuration or safety limits (Fail-Safe Behavior Section 13.1.3; Device Classes Section 7.8), which remain independent layers of protection.
5. **Limits of temporary states.** `revertAfter` ends temporary output states and temporary setpoints when the controller fails, and latched stops keep the stops of interlocks and emergency modes, within the limits of Sections 11.7 and 17.5. Other states without an end, such as a power-loss mode's, and setpoints with no known end, are protected after a controller failure only by fail-safe behavior and the devices' own limits.
6. **Privacy.** A configuration can reveal where its owner lives (`astronomy.location`), when they are at home (schedules, feed modes, overrides), and which devices they own (bindings). A location is needed only for sun and moon times: owners **SHOULD** use the location of the biotope rather than their own, or round it to one decimal place, about 11 km. Controllers **SHOULD** remove bindings and state, and offer to remove the location, names, and notes, before a configuration is shared (Section 19.3).
7. **Resource use.** A very large configuration could keep a controller from evaluating every second. A controller **MAY** reject a configuration that exceeds documented limits, such as the number of elements, and **MUST** then say which limit it exceeds.

Appendix A — Approval Checklist

Complete this checklist before requesting Working Group approval. Link each item to the evidence.

Item	Status	Details
Section 2 describes compliance with all five tenets.	Done	
A public review period of at least 30 days has been announced and completed.	Not started	
At least two independent implementations exist, at least one from an organization not represented among the Authors.	Not started	
The implementations have exchanged configurations, and made the same decisions from the same inputs.	Not started	
Every normative reference is publicly available.	Done	
The schema, presets, and examples are published, and the examples pass the validator.	Done	
The implementations have passed the test procedures in Section 20.4.	Not started	
The Open Issues appendix is empty, or each open issue is explicitly deferred.	Not started	
No unresolved Exclusion Notices for this Specification are recorded in <code>Notices.md</code> .	Done	
The Maintainer has recorded consensus to approve.	Not started	

Appendix B — Examples (Informative)

The `examples` directory contains complete configurations:

- `reef-system.json`: a reef system with a heater and a chiller in one function with a dead band, acting on the higher of two temperature sensors; a return pump that slows at night, through a virtual output; a skimmer, ozone on ORP, and a kalkwasser reactor, with their interlocks; a cooling fan under proportional control; a moonlight that follows the moon; two feed modes that both run the feeder, one started by the owner and one every evening; maintenance, water-change, leak emergency, and battery modes; bindings; and an active override.
- `quarantine-treatment.json`: a quarantine tank with a raised-temperature treatment program, which is running, and a program that slowly lowers salinity with fresh water, with a latching overflow interlock.
- `planted-co2.json`: a freshwater planted tank, configured in °F and dKH, with a seasonal temperature and a temporary treatment offset, a photoperiod from a sun table, CO2 on pH that stops outside the photoperiod and while the owner's last carbonate hardness test was low, and a weekly water change. It has no bindings or state, so it can be shared.
- `minimal.json`: one heater and one sensor.

Appendix C — Commercial Programming Models (Informative)

How the programming models of commercial controllers correspond to this Specification, as a guide for importers. Each row describes the model of one of four widely used commercial controller ecosystems.

Table 31 — Commercial programming models

Approach	Programming model	In this Specification
Text programs	A text program per output, in which the last true line wins	One function per output for the base behavior; the exceptions that later lines express become modes, interlocks, and overrides, in the order of Table 25. Virtual outputs and feed modes map directly.
Logic gates	Functions assigned to hardware; logic gates, latches, and timed gates	Functions and outputs, linked by bindings; conditions of Section 10.
Dependencies and combiners	Outputs with inputs, dependencies, and combiners	<code>requires</code> interlocks for dependencies; <code>all</code> and <code>any</code> for combiners; derived inputs for averaged and higher-of-two sensors.
Modes per output	Always on, off, schedule, or sensor, per output	<code>fixed</code> , <code>condition</code> with a <code>schedule</code> , and <code>setpoint</code> functions.

Appendix D — Open Issues (Informative)

Issues 1 to 29 were open in revision 0.1. The Project Lead decided issues 30 and 31 on 15 October 2026, and issues 32 and 33 on 16 October 2026, accepting the recommendation in each. On 17 October 2026 the Project Lead froze the scope of revision 1.0, and decided issues 1, 3, 4, 9, 13, and 17; revision 0.3 applies the decisions, and resolves issues 18 and 23 to 27, which other specifications settled. Issue 34 records the choices that revision 0.3 made where those decisions left them open, which the Project Lead accepted on 18 October 2026. Issues 2, 7, 12, 14, 15, 19, and 29 are deferred beyond revision 1.0, and are tracked outside this Specification. Issue 16 stays open until the bench tests and owners' experience confirm its numbers. Each issue says how a revision resolves it, or why it stays open. Issue numbers do not change, because other specifications refer to them.

1. **Default precedence.** *Resolved in revision 0.3, as the Project Lead decided on 17 October 2026: Table 25, as it stands.* Other orders were defensible: overrides above the emergency and power-loss modes, so that an owner who is present can run equipment during a leak or on battery; or the ordinary modes above overrides, so that pressing "feed" always pauses the pumps. Interlocks, `stop-all`, and the emergency and power-loss modes stay above a person's override, because an owner who is present can end an emergency mode; overrides stay above the ordinary modes, so that an owner can run a pump during a feed. Sections 14.1 and 17.3 no longer call the order provisional.

2. **How far presets go.** *Resolved in revision 0.3, as the Project Lead decided on 17 October 2026.* Presets give structure and equipment-protection values, but no target values or limits (Section 18.1). Targets stay the owner's own, entered or confirmed by the owner; the alternative was editable starting points by system type, which are husbandry advice. Section 18.1.2 is no longer provisional.
3. **Fans.** *Resolved in revision 0.3, as the Project Lead decided on 17 October 2026, with Data Model revision 0.4 and Fail-Safe Behavior revision 0.4.* The Data Model adds the equipment category `fan`, and Fail-Safe Behavior recommends for it `run` on loss of contact, `fail-safe` after power-up, no power-on delay, and `stop` on battery. The fan preset now uses `fan` (Tables 28 and 29).
4. **Devices running their own programs.** *Resolved in revision 0.2.* Device Classes revision 0.2 defines `runProgram` for lights and pumps, and `set` with `enabled` for dosing heads and analyzers, and says how to resume control without changing what an endpoint does (its Section 7.13). A controller now returns an endpoint that was following its program to the program when the last layer with a demand on it ends (Section 17.3.4), and resumes control of an endpoint with no demand without taking it off its program (Section 17.4.11).
5. **Pending reverts and power-up.** *Resolved by Device Classes revision 0.2 and Fail-Safe Behavior revision 0.2.* With `powerOnBehavior last`, an endpoint restores a pending revert's target, not the temporary state [OAP-DC Section 7.4.8; OAP-FS Section 9.1].
6. **Power loss and interlocks.** *Resolved by Fail-Safe Behavior revision 0.2.* This Specification applies interlocks in power-loss mode (Section 14.6), and Fail-Safe Behavior Section 9.5.2 now says so too.
7. **Device setpoints have no revert.** *Resolved in revision 0.3, as the Project Lead decided on 17 October 2026.* A setpoint written to a self-regulating heater or chiller outlasted a failed controller, even a temporary one from an offset or a step program. A temporary setpoint, whose end is known, from an offset with an end or a step other than a step program's last, is now sent with `set`, its `setpoint`, and `revertAfter`, so that it ends when the controller fails; a setpoint with no known end is written without, and stays on the device (Sections 11.7.6 and 11.7.7).
8. **Binding identifiers.** *Resolved by the Device Protocol specification.* It defines the identifiers that follow `device-protocol:` (its Section 7.3). A device's key ID changes when it is factory reset, so its bindings must then be made again; that specification asks controllers to offer to move them.
9. **Data Model configuration.** *Resolved by Data Model revision 0.2.* Configuration moves in configuration documents rather than Data Model documents, and history records modes, overrides, and step programs as events of the types `mode`, `override`, and `program`, whose `element` is the element's `id` (its Section 8.6). Owners' target ranges are Data Model Ranges (its Section 8.9).
10. **Conformance levels.** *Resolved in revision 0.3, as the Project Lead decided on 17 October 2026: one level.* The question was whether small controllers could omit features, such as moon times or step programs, under a declared conformance level. There is no reduced level in revision 1.0: a controller conforms when it implements every requirement that applies to it, as lights and pumps that run their own programs already must (Section 20.2).
11. **Timing figures.** *Open: to be confirmed on the breadboard controller and with owners' experience.* The evaluation interval of 1 s, the tolerances, the expiries proposed for overrides, and the preset values need confirming on the breadboard controller.
12. **States without an end after controller failure.** *Resolved in revision 0.3, as the Project Lead decided on 17 October 2026.* An emergency mode's stops, power-loss states, the output actions of a program's last step, and the actions of a mode of an extension kind that ends only by `end.when` have no known end, so they have no `revertAfter`, and if the controller failed, each device applied its fail-safe behavior, which for a return pump is `run`. A stop with no known end that an interlock or an emergency mode sends is now latched, as Device Classes revision 0.5 defines, so a return pump stopped for a leak stays stopped when the controller fails, and the alarm output, which runs on loss of contact, calls a person (Section 17.5.4) [OAP-FS Section 8.4]. Other states without an end, such as a power-loss mode's or a step program's last step, keep the fail-safe behavior they have (Section 17.5.6).
13. **Faults and alarms.** *Resolved in revision 0.3, with Alarms and Notifications revision 0.3.* The Alarms and Notifications specification gives each code of Table 27 a default severity (its Table 8), and defines an alarm function that switches alarm outputs (its Section 15). Table 14 now lists `alarm` among the function

- types, in place of the provisional extension name `org.openaquariumproject.alarm` (Section 11.10), and modes and `stop-all` leave alarm outputs to their function (Sections 14.4.4 and 17.3.1).
14. **Moon accuracy.** *Resolved in revision 0.2.* Section 10.7.5 measures moon events against the method `oap-1` of the Lighting Programs specification, which that specification checked against an independent ephemeris (its Section 9.6). The figures stay provisional (issue 16).
 15. **Reporting rates.** *Resolved in revision 0.2.* The default maximum ages of Table 10 follow each sensor's `measurementInterval` and the reporting interval of its binding, as Fail-Safe Behavior Section 10.2.3 recommends, so a battery-powered sensor that reports every 3600 s gets a default of three hours. A controller warns when a configured `maxAge` is shorter than three of those intervals (Section 9.8.9).
 16. **Revert targets.** *Resolved in revision 0.2.* Device Classes revision 0.2 keeps a pending revert's target when another command with `revertAfter` arrives (its Section 7.4.6), and Section 17.5 now relies on it, for endpoints of revision 0.2 or later.
 17. **Water level function types.** *Resolved in revision 0.3, with Water Level and Water Change revision 0.3.* The `Water Level and Water Change` specification defines top-off, water change, and RO/DI functions, which it carried as provisional extension types, with their saved state under `state.extensions`. Revision 0.3 lists `top-off`, `water-change`, and `ro-di` among the function types (Table 14), keeps each function's saved state as `state.functions["<function id>"]` (Section 17.8), puts the checks those functions apply to every run of their outputs at the precedence of interlocks (Section 11.10.6 and Table 25), says that a water change sends `dose` to the dosing heads it drives (Section 8.3.1), describes the water-change mode that a water change follows and lets water-change modes of different systems be active together (Sections 11.10.7 and 14.3.3), allows the stops and holds that those functions send without `revertAfter` (Section 11.10.8), and points the `auto-top-off`, `water-changer`, and `ro-di` presets to the new types. Raised by the Water Level and Water Change specification (Appendix D, issue 8).
 18. **Dosing function type.** *Resolved in revision 0.3, with Dosing revision 0.3.* The `Dosing` specification defines a dosing function, which it carried as a provisional extension type. Revision 0.3 lists `dosing` among the function types (Table 14); says that a dosing function sends `dose` to `doser` outputs, which have no running state, only while no higher layer holds them (Section 8.3.1); forbids `prime`, `calibrate`, and `completeCalibration` as mode commands, because only a person starts them (Section 14.5.2); and points the `doser` preset to the new type. Whether a dose held by a mode is given later is the Dosing specification's (Table 29). Raised by the Dosing specification (Appendix D, issue 8).
 19. **Water testing function type.** *Resolved in revision 0.3, with Water Testing revision 0.3.* The `Water Testing` specification defines a testing function, of the type `water-testing`. Revision 0.3 lists it among the function types (Table 14); allows a testing function with no outputs (Section 11.10.9); has an `endpoint` input of an analyzer that a testing function drives read that function's accepted, corrected results (Section 9.2.3); notes that its demand on its analyzers is stop while it delays or suspends testing (Section 11.10.9); points the `analyzer` preset to the new type; and extends the preset's water-change stop by 1800 s, while new water mixes, as the Water Testing specification advises (Table 29). Raised by the Water Testing specification (Appendix D, issue 8).
 20. **Lighting function type and astronomy.** *Resolved in revision 0.3, with Lighting Programs revision 0.3.* The `Lighting Programs` specification defines a lighting function, which it carried as a provisional extension type, and an astronomical method, `oap-1`. Revision 0.3 lists `lighting` among the function types (Table 14), describes its demand on a light, the program, and that it returns a light to its program when a higher layer's demand ends (Section 8.3.2 and Table 25), and points the `light` preset to the new type. Section 10.7 now computes sun and moon events by the method `oap-1`, takes its solar day for a shifted location, adds its `noonAt` and `shiftDays` to `astronomy`, limits latitudes to 60°, and asks for sun events within 2 s of it, so that schedules and lights agree to the second. Raised by the Lighting Programs specification (Appendix D, issues 2 and 6).
 21. **Flow function type.** *Resolved in revision 0.3, with Flow Programs revision 0.3.* The `Flow Programs` specification defines a flow function, which it carried as a provisional extension type, and which keeps no saved state. Revision 0.3 lists `flow` among the function types (Table 14), describes its demand on a pump, the program, and that it returns a pump to its program when a higher layer's demand ends (Section 8.3.2 and Table 25), points the `circulation-pump` preset to the new type, and lets the `return-pump` preset use it for constant flow (Table 28). Storms stay the extension kind `org.openaquariumproject.storm` of that

specification; revision 1.0 adds no standard mode kinds. That specification follows modes with triggers, so a mode that only a flow function's trigger follows changes no output of its own. Raised by the Flow Programs specification (Appendix D, issue 2).

22. **Calibration holds.** *Resolved in revision 0.2.* The [Probe Interface](#) specification keeps a probe's inputs unavailable during a calibration, and after it until a person confirms that the probe is back and its readings have settled, and after a probe is reconnected (its Section 14.8). Section 9.8.6 names this among the causes of an unavailable input, so that a controller can show why a function stopped, and revision 0.3 names the other holds of that specification as well. Raised by the Probe Interface specification (Appendix D, issue 10).

Decided by the Project Lead

30. **Modes that end.** *Resolved in revision 0.2.* Every mode has a [duration](#) or an [end.when](#), including modes of extension kinds (Section 14.1). The schema and the validator enforce it.
31. **Safety functions on radio.** *Resolved in revision 0.2; extended in revision 0.3 by the Project Lead's decision of 17 October 2026.* A controller checks each element that carries a safety function for a path that uses no radio link, as Fail-Safe Behavior Section 13.4 defines, warns its owners persistently while one has none, raises [safety-path-missing](#), and does not refuse the configuration (Section 13.7). On 17 October 2026 the Project Lead decided that an output that stops by itself on loss of contact gives a path only while its [failSafeTimeout](#) is at most 300 s (provisional), and that a function with several inputs, such as leak response, is judged as Fail-Safe Behavior Section 13.4.3 judges it; revision 0.3 applies both, and judges a system's leak response as one safety function (Section 13.7).

The following issues arose in revision 0.2:

32. **Starting modes of every kind.** *Decided by the Project Lead on 16 October 2026; resolved in revision 0.2.* Section 21, item 3, needs [start-mode](#) to start modes of every kind: emergency modes and modes of extension kinds, as well as feed, maintenance, and water-change modes. Security and Access Table 6 described [start-mode](#) for the last three only. The Project Lead decided to widen it rather than add operations: it now starts a mode of any kind, ends one other than an emergency mode, and starts or ends a lighting scene, as the Lighting Programs, Flow Programs, and Matter Mapping specifications already use it.
33. **A lost heater that keeps its state.** *Decided by the Project Lead on 16 October 2026; resolved in revision 0.2: kept.* While the device of a heater is lost, its state is unknown for its [failSafeTimeout](#) plus 2 s, and an exclusive interlock holds the chiller stopped meanwhile (Section 13.4.3). After that, a heater whose [failSafeAction](#) is [continue](#), and that was last commanded to run, counts as running (Section 9.3.3), so the interlock holds the chiller stopped until the device reports again, however hot the water becomes. The heater may be running, so starting the chiller could set the two against each other, which is what the interlock prevents. The Project Lead decided to keep this, as recommended, because the chiller's own protection and the range alarms of the Alarms and Notifications specification still watch the water, and a heater and a chiller that both regulate themselves can use one [device-setpoint](#) function instead, which needs no exclusive interlock (Section 13.5.3). Revision 0.3 counts a pending revert that falls due meanwhile, as Fail-Safe Behavior Section 7.3.2 keeps it (Section 9.3.3).

The following issue records the choices of revision 0.3:

34. **Choices of revision 0.3.** *Resolved in revision 0.3: the Project Lead accepted each choice on 18 October 2026.* Revision 0.3 made these choices where the decisions it applies left them open:
- the stop of a [requires](#) interlock while the output it requires is paused has a known end, so it is sent with [revertAfter](#) for the pause's end plus the interlock's [delay](#), rather than latched, and the dependent output restarts after the one it requires even on a failed controller's devices (Section 17.5.5);
 - a temporary setpoint is sent with [set](#) only while its output's resolved state is run, so that it never starts a heater that another layer holds stopped, and is otherwise written to the attribute (Section 11.7.6);
 - a mode MUST NOT list an output that an alarm function drives, which the validator checks, rather than list it to no effect (Section 14.4.4);

- the stops that reach equipment under local control are all the stops that protect the system: those of the first layer of Table 25, other specifications' checks among them, those of the emergency and power-loss modes, and a heater's or chiller's own cutoff (Section 17.4.2);
- a system's leak response is one safety function, which has a path for an output when one of its interlocks or emergency modes stops the output with one, as Fail-Safe Behavior's example and the Water Level and Water Change specification judge it (Section 13.7.3);
- mode commands that only a person starts are a dosing head's `prime`, `calibrate`, and `completeCalibration`, and a top-off unit's `resetFault` (Section 14.5.2);
- sun events within 2 s of the method `oap-1`, moonrise and moonset within 60 s, and moon illumination within 0.1 percentage points, all provisional, with latitudes from 60° S to 60° N (Section 10.7.5);
- the `analyzer` preset extends its water-change stop by 1800 s, while new water mixes (Table 29); and
- the example's sump fan has the equipment category `fan`, and the Matter Mapping specification's example and its Appendix B follow it, so that home apps cannot stop the fan there (Appendix B).