Print copy. This is the print copy of an article whose version of record is a web page. That page, with the data and code behind every figure, is at
https://hertzforge.com/journal/2026-006/v1/
DOI of this version: 10.5281/zenodo.24260838

SYSTEM DESCRIPTION

An Open Vibration Controller Anyone Can Change: How Hertzforge Is Built, and Why

Kevin Cross ORCID 0009-0002-6608-9426
Brandon Zwink ORCID 0000-0003-1341-2293
The authors of this article are the journal's editors. Its reviewers are drawn from outside the author list.

Evidence	**Bench.** The Hertzforge software on its main branch at commit b587d5a1, its automated tests and its written rules; loopback runs on one NI cDAQ-9178 chassis with an NI 9263 output module wired to an NI 9234 input module, timed with a bench multimeter; simulator runs.
Version	Version 1, published 2026-11-15. First published 2026-11-15. Change note: First published.
Latest version	10.5281/zenodo.24260837
Cite as	Cross, K., and Zwink, B. (2026). An Open Vibration Controller Anyone Can Change: How Hertzforge Is Built, and Why. Hertzforge Journal 1, 2026-006, version 1. https://doi.org/10.5281/zenodo.24260838

Abstract

Hertzforge is an open-source program for structural dynamics testing that joins a vibration controller with an analysis program. Anyone may change any part of it, including the math that decides each shaker drive, so it has to be built so that a change cannot remove what protects the shaker and the article on it. This article describes how it is built and why. Three programs in three languages split the work: a Rust control service that alone drives hardware and holds every protection, a Python analysis service, and a TypeScript shell that draws and does no math, joined by one versioned protocol. A control law is a sandboxed WebAssembly file: it proposes the next drive, and the controller checks and limits what is played. Hardware sits behind an abstraction in which a backend states what it can do and what it has proven, and on real hardware a closed-loop test refuses to arm until the input and output are shown on a bench to share one clock. A simulator is one of the backends, so the same test code runs on simulated structures and on hardware, and loopback runs showed what the simulator hides. Most of these decisions were made for Eigen, an earlier controller that was never finished as a product and became Hertzforge's control side; the article says which carried over, which changed, and the defects that shaped them. The evidence is the code, its tests and loopback runs on one NI chassis.

Contents

Summary 3

1. What This Article Covers 3
1.1 Terms 3
1.2 Where the Program Came From 4
1.3 What the Program Is For 4
2. Three Programs, One Protocol 5
2.1 Who Owns What 5
2.2 Why Three Languages 6
2.3 One Protocol 6
2.4 The Service Says How to Draw It 6
2.5 One Controller, Any Number of Observers 7
3. A Control Law Is a File 7
3.1 What a Law Is 7
3.2 Why WebAssembly 7
3.3 The Law Proposes, the Controller Decides 9
3.4 Declared Parameters 10
3.5 Checked Before Hardware, Recorded Every Run 10
4. The Drive Path 10
4.1 What It Is 10
4.2 The Invariants 10
4.3 The Safety Tests and the Rules for a Change 11
5. The Hardware Abstraction: What a Backend Has Proven 11
5.1 Who Closes the Loop 11
5.2 One Resolver, and No Silent Downgrades 12
5.3 A Capability Is a Claim of Proof 12
5.4 The Clock Must Be Proven, Not Assumed 12
5.5 The Abort Must Be Fast Enough, and Measured 13
5.6 A Defect That Shaped the Rules 13
6. The Simulator Is a Backend, and What It Hides 14
6.1 One Code Path 14
6.2 What the Bench Found That the Simulator Did Not 14
7. A Run Is Kept for Analysis 14
8. What Carried Over From Eigen, and What Changed 15
9. Limits 16
Contribution Statement 16
How This Was Made 16
Acknowledgments 17
References 17
Implemented In 17
Version History 17

Summary

A vibration controller drives the shakers in a test and adjusts the drive, loop by loop, until the accelerometers on the test article read their targets. The math that makes that adjustment is called the control law. In commercial controllers it is fixed and closed. Hertzforge lets anyone read it, change it, or load a law of their own as a single file. That is the point of the project, and it raises a hard question: if anyone can change the math that drives a shaker, what stops a bad change from breaking the shaker, the fixture or the article?

Hertzforge's answer is to put every protection below the control law, in code the law cannot reach, and to hold that code to stricter rules than the rest. This article describes the parts of the program that make that work, and the reasons for each:

- **Three programs, one protocol.** A Rust control service is the only program that drives hardware, and it holds every protection. A Python analysis service does the offline analysis, carried over from Visual Dynamics. A TypeScript shell draws everything and computes nothing. They speak one versioned protocol, tested against the same sample messages in all three languages.
- **The law proposes, the controller decides.** A control law is a WebAssembly file run in a sandbox. It can compute and nothing else. The controller checks what it returns, may reduce it, never makes it more aggressive, and records which law ran.
- **A written drive path.** The code that can change what reaches an output, or how a run stops, is listed by file. It has seven invariants and a list of safety tests that the automated checks make sure still exist and still run.
- **Hardware states what it has proven.** A backend declares what it can do. On real hardware a closed-loop test refuses to arm until the input and output are shown, on a bench, to run on one clock.
- **The simulator is a backend.** The same test code runs against simulated structures and against hardware. Loopback runs on hardware found effects the simulator does not have, and one default chosen on the simulator was wrong on hardware.
- **A run is kept for analysis.** Every control module writes a run folder that the analysis side turns into an ordinary project, labeled as simulated or hardware.

Most of these decisions were first made for Eigen, an earlier controller meant to be a commercial product. Eigen was never finished as a product; it became Hertzforge's control side. The article says which carried over, which changed when the work became open, and the defects found in Eigen that shaped them.

The evidence is the code at one commit, its tests, and loopback runs on one NI chassis in which the output is wired back to the input. Two backends exist: the simulator and NI.

1. What This Article Covers

1.1 Terms

A **shaker** is an electrodynamic actuator that shakes a **test article** (the item under test) through a fixture. A **drive** is the voltage signal sent to the shaker's amplifier. **Control accelerometers** measure the motion the test is judged on; **limit accelerometers** measure motion that must stay under a ceiling. Once per **loop**, the controller compares what was measured with the target and computes the next drive. The math that does that is the **control law**. Everything else around it (acquisition, spectral estimates, drive synthesis, protection and records) is the **core**.

An **abort** takes the drive to zero at once. A **controlled stop** ramps the drive down. A **backend** is the code that talks to one kind of hardware. The **frequency response function (FRF)** relates each drive to each measured response; the controller estimates it before a test and uses it to compute drives.

1.2 Where the Program Came From

Hertzforge joins two programs. **Eigen** was a vibration controller in Rust that Kevin Cross began in July 2026, intended as a closed-source commercial product. Its design record was settled that month. Eigen was never finished as a product and was never released on its own. **Visual Dynamics** is a structural dynamics analysis program in Python by Brandon Zwink (Zwink 2026). At the end of October 2026 the authors decided to release Eigen's controller under the MIT license and combine the two into one open program, and Eigen's development ended there. Its controller was imported into Hertzforge on 1 November 2026 under new names, and continues as Hertzforge's control side; the same day the authors agreed that Visual Dynamics' analysis code would move into Hertzforge and its development continue there.

The architecture described here is Hertzforge's on its main branch at commit `b587d5a1` (15 November 2026). Where a decision came from Eigen, the article says so and gives the reason it was made then. Where the decision changed, it says why.

1.3 What the Program Is For

Hertzforge is meant to be a platform on which the test community can try new control ideas, which closed controllers do not allow. That aim sets the two requirements the rest of the article serves:

1. **Anyone can change the control law,** and a law can be loaded from a file without rebuilding the program.
2. **No change to a law can remove a protection.** A law that is wrong may give a poor test. It must not be able to give an unsafe one.

The program is offered for research and use at the user's own risk. The second requirement is what makes it reasonable to ship a law that works on most tests and fails on some, with its limits stated, instead of holding it back until it is perfect.

2. Three Programs, One Protocol

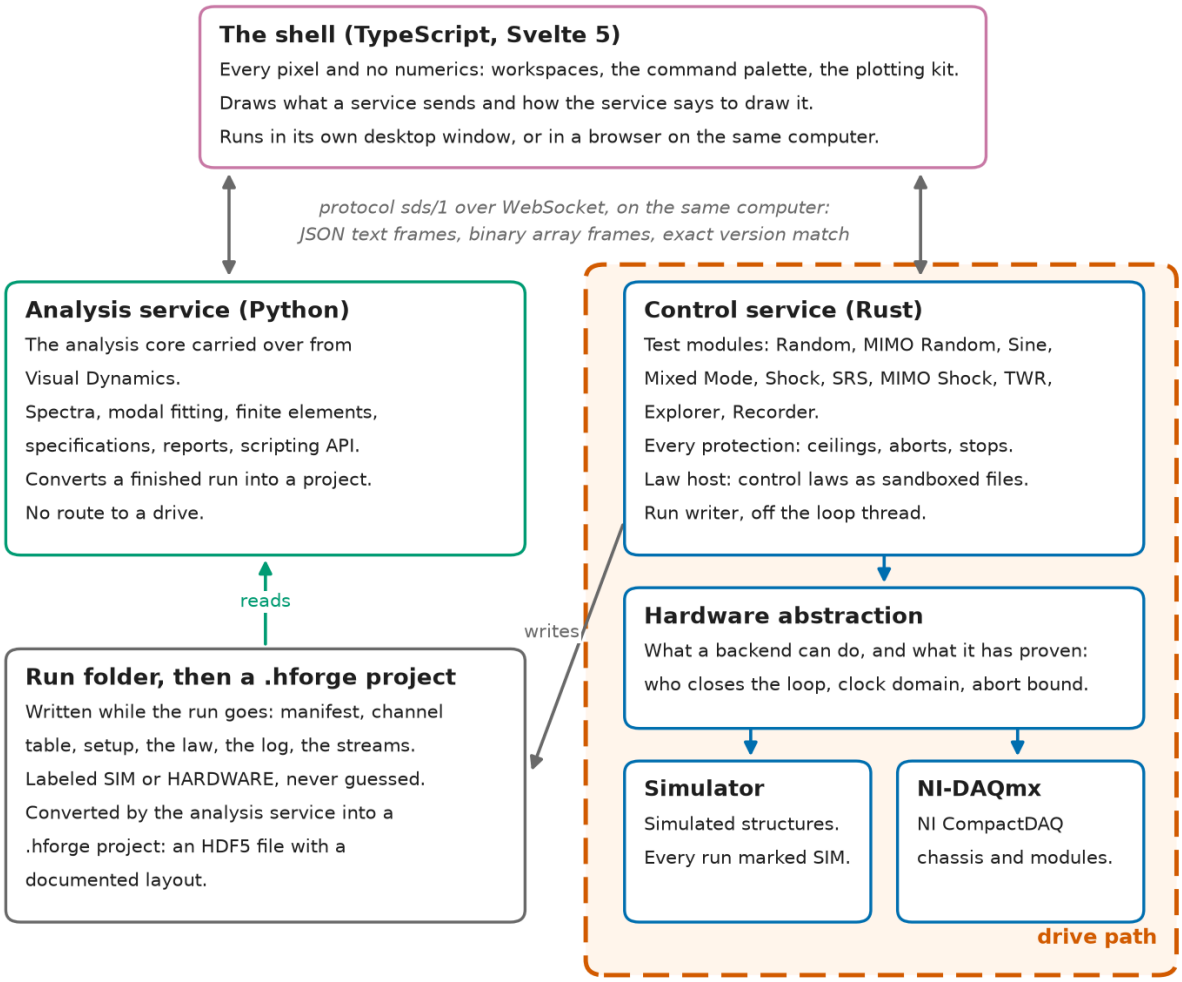


Figure 1 shows the parts. Hertzforge is three programs, each in one language, each with one job.

2.1 Who Owns What

The repository states the split as rules (Cross and Zwink 2026, README, “Layer Rules”):

- **Rust owns anything the real-time loop or the simulator runs,** the hardware backends, and the numerical kernels that control and analysis share.
- **Python owns everything offline:** analysis, modal fitting, finite elements, specification tools, reports and the scripting interface.
- **TypeScript owns every pixel and no numerics.** A view asks a service for a result and draws it.
- **The control loop and the abort path never depend on the shell, the network or Python.**

The control service is the only program that drives. The analysis service has no route to an output channel, and the shell is a client: it may show a limit or grey out a button, but no protection rests on it (Section 4.2, invariants 4 and 5).

2.2 Why Three Languages

Rust for the drive. Eigen chose Rust for its control core in July 2026 for two reasons that still hold. A program that drives a shaker must not pause at random, so a language with a garbage collector was ruled out for the loop. And the code was to be written largely by AI coding agents under one engineer's direction; Rust's compiler rejects whole classes of concurrency and memory errors before the code runs, which made it a reviewer on exactly the code that drives a shaker.

Python for analysis. When the two programs were combined at the end of October 2026, five ways were weighed. Rewriting Visual Dynamics in Rust would have thrown away its scripting interface, the scientific Python libraries it rests on and the test suite that checks its results. Moving Eigen's interface into Visual Dynamics' Qt window would have kept a single-window layout that was already crowding its features, and would have put heavy analysis and live display on one thread. Keeping two programs joined by files would not have made one program. The choice was to keep Visual Dynamics' analysis core as a service, unchanged at first, keep Eigen's control core as the other service, and replace both user interfaces with one. The analysis core had no dependence on its window at load time, which is what made this possible. The arrangement has the same shape as JupyterLab, a TypeScript front end talking to a Python process over a socket (Project Jupyter 2026).

TypeScript for the screen. Eigen chose a web-technology interface, Svelte inside a Tauri desktop window, over Qt in July 2026. The reasons were the license (Qt's charting components are GPLv3 or commercial), the size and age of the body of code AI coding agents have learned from (web interfaces are the largest), and the ceiling on visual quality. The shell that Hertzforge uses grew from Eigen's. It runs in Tauri as a desktop application, or in a browser on the same computer.

The cost of three languages is keeping them in step. The protocol is how that is done.

2.3 One Protocol

The three programs speak one protocol, `sds/1`, over WebSocket (Cross and Zwink 2026, `schema/PROTOCOL.md`; `sds` was the project's working name). Text frames carry JSON messages with no arrays. Binary frames carry arrays: a four-byte magic number, a JSON header, and the arrays laid out little-endian, each starting on an eight-byte boundary, so a decoder in any of the three languages can view them without copying. On connecting, the server sends a greeting naming the protocol version, and the client closes the connection on any difference. There is no graceful degradation across versions, so two programs that disagree about a message never run a test together.

A folder of **golden frames** holds sample messages, text and binary, with the decoded result each should give. The Rust, Python and TypeScript test suites each read the same files, so a change to the protocol that one language misses fails its tests.

2.4 The Service Says How to Draw It

Every result a service sends carries the views to draw it with: the axes, their units and scales, and each series with a role (measured, reference, tolerance, abort, limit, drive) that sets its line style, so no meaning rests on color alone. A spectrum computed as a density per frequency line is drawn flat across each line's bin, as a staircase, and the service that computed it sends the bin edges. This rule came from Visual Dynamics. It keeps the shell from inventing numbers by joining points with lines the computation never made.

The same idea runs through the analysis side. The analysis service describes each operation it offers (its parameters with types, units and limits, and the views it returns), and the shell builds the panel from the description. A new analysis feature that fits this pattern needs no TypeScript. Section 4.4 shows the same pattern for control laws.

2.5 One Controller, Any Number of Observers

Several screens may connect to the control service at once. One connection at a time holds control and may change what runs; the others observe. In the Random modules, stopping is open to every connection: any screen watching a test that goes wrong can stop or abort it, because those requests can only end a run. If the controlling screen disappears without saying goodbye (a dropped network, a sleeping computer), the run continues for 30 seconds while the seat is held; if no one claims it, the service ends the run with the controlled stop, never with an abort (`schema/PROTOCOL.md`, "Controller and Observers").

The control service listens only on the computer it runs on, and the application makes no network requests beyond its own services, because test laboratories often run on closed networks. The reasoning goes back to Eigen, which settled in August 2026 that a controller would run on a closed, dedicated network rather than on a site's general network, because a controller should not depend on a network its makers do not control. Hertzforge's control service goes further and accepts connections only from its own computer.

3. A Control Law Is a File

3.1 What a Law Is

In Hertzforge a control law is a **WebAssembly component**, a compiled file in a portable format that a host program runs in a sandbox (Bytecode Alliance 2026a). Its interface is written in WIT, WebAssembly's interface language, one interface per test type. At commit `b587d5a1` the repository defines seven: single-output Random, MIMO Random, classical shock, SRS shock, MIMO shock, single-output Sine and time waveform replication (`wit/control-law/`).

At this commit, the operator can choose a law file for MIMO Random, Classical Shock, SRS, MIMO Shock, Sine and TWR, and for the parts of a Mixed Mode test. The Random module itself runs its built-in law; a single-output Random law file runs as the random floor of a Mixed Mode test.

A law has a manifest (its name, version, author, summary, parameters and diagnostics), a constructor called once per run, a call that returns the first drive after the system has been identified, and a call that returns the next drive each loop. It keeps its own state between calls, and nothing carries over between runs (Cross and Zwink 2026, `docs/control-law-spec.md`).

3.2 Why WebAssembly

The authors considered five ways for a user to supply a law. Table 1 compares them.

Table 1: Ways to supply a control law, as compared on 2 November 2026. "One file" means one law file runs on every platform; "Repeatable" means the same inputs give the same drive, which is what lets a recorded run be replayed.

Way	If the law crashes	One file	Repeatable
Compiled into the program	Takes the program down	No	Yes
Native library	Takes the program down, with the drive	No	Yes
WebAssembly component	Returns an error; the core holds	Yes	Yes
Python inside the control service	Usually an error; native crashes possible	No	Mostly
Python in a separate process	Contained	No	Mostly

WebAssembly was chosen for four reasons. A law that divides by zero, runs out of memory or loops forever produces an error or hits its deadline, and the core holds the last good drive; it cannot corrupt the controller's memory. The same file runs on macOS, Windows and Linux, so a law is built once and shared as one file. The host gives a law no files, no network, no environment, a clock that does not move and fixed random seeds, so a shared law cannot read or send anything and gives the same output for the same input. And the runtime used, Wasmtime, has a permissive license (Bytecode Alliance 2026b).

Python was not chosen for live laws, because it would put a large runtime and a native crash path in the program that holds the drive. It remains useful for prototyping: where a module records every call of its law, a Python version of a law could be run against a recorded run and compared with what ran. That tool is planned, not built.

Rust is the supported language for writing laws, because the core and its math are Rust, and the repository publishes a crate with the interface bindings and helpers. Any language that builds a WebAssembly component against the WIT files can produce a law file.

History. Eigen had a plug-in boundary too, for a different reason. Its test modules were to be native libraries behind a stable interface that doubled as a license boundary, so that modules could be sold separately. Eigen's own design record admitted that a native library loaded into the controller could crash it, and made a separate process the default for any module the controller had not compiled. When the controller became open, the licensing purpose went away. What remained was the need to run someone else's code next to the drive, and the WebAssembly sandbox meets it more simply than a separate process.

Prior art. Rattlesnake, the open-source vibration controller from Sandia National Laboratories, loads user control laws for MIMO random vibration at run time as Python functions, generators or classes, which receive the specification, the FRF estimate and related quantities and return a drive matrix per frequency line (Rohe, Schultz and Hunter 2021; Sandia National Laboratories 2026). Hertzforge's MIMO law inputs are close to that set. It differs in three ways: parameters are typed and declared rather than passed as a string the law parses, protection is held by the controller rather than left to the law, and the law's identity is recorded with the data. No Rattlesnake code was read or used in building Hertzforge; the concepts were taken from its published documentation and are credited here and in the repository's acknowledgments.

3.3 The Law Proposes, the Controller Decides

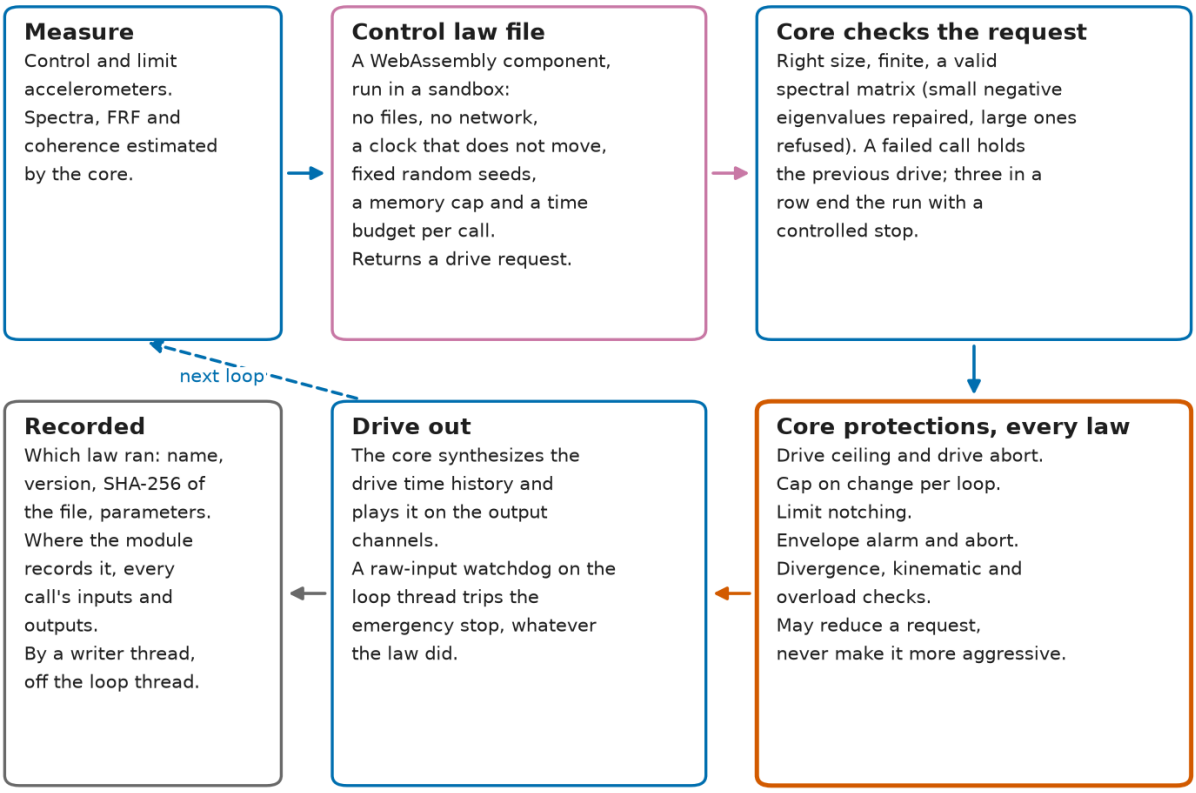


Figure 2 shows one loop. A law only computes the next drive request. Everything else is the core's, and applies to every law, built in or loaded from a file (`docs/control-law-spec.md`, section 3):

- **The core checks every output.** It must have the stated dimensions and be finite; a single-output drive spectrum must not be negative; a MIMO drive spectral density matrix must be Hermitian and positive semidefinite. A negative eigenvalue between 10^{-6} and 10^{-3} of the largest in size is set to zero and the requested drive levels restored; a larger one fails the call. Lines outside the controlled band must be zero.
- **A failed call holds the previous drive.** Three failed calls in a row end the run with a controlled stop, recorded with the reason.
- **The core keeps every protection:** the drive ceiling and drive abort, the cap on how far the drive may change in one loop, limit notching, the envelope alarm and abort, and the divergence, kinematic and overload checks. It may reduce a law's request to satisfy one. It never makes a request more aggressive than the law asked for.
- **A law's diagnostics can add a trip but never suppress one.** A law that reports nothing is no less safe than one that reports everything.
- **Budgets.** A law has a memory cap (256 MiB by default) and a time budget for each call (a quarter of one loop's acquisition time by default). Exceeding either is a failed call.

The built-in laws go through the same door as imported ones. The rule the authors set when the design was chosen was that if the default law could not be written as a plug-in, the interface was wrong.

3.4 Declared Parameters

A law declares its parameters (a number, an integer, a toggle, a choice, a curve against frequency, or one value per control channel), each with its default and, where it applies, its unit and range. The shell draws the input boxes from the declaration, and the core checks every value against it before the law is built. A law ships no user interface code and parses no free-form text. This is the same pattern as the analysis operations of Section 2.4: the code that knows what an input means declares it, and the shell draws it.

3.5 Checked Before Hardware, Recorded Every Run

A law file is checked before it may drive hardware. `law check` runs it through whole simulated runs, under the core's protections, on a fixed set of simulated test articles for its test type: for MIMO Random, well-conditioned and ill-conditioned multi-shaker models, a rigid plate, measurement noise and a level ladder; for Classical Shock seven articles, for SRS six and for Sine seven. A pass is recorded against the SHA-256 of the file. The simulator accepts any law. On a hardware station, a module refuses to arm with a law file that has not passed for its exact bytes (`law_not_checked`). The built-in laws are cleared without the check.

Every run folder names the law in use. How much more is kept differs by module at this commit. MIMO Random, MIMO Shock and TWR record every call of the law, with what it was given and what it returned; Classical Shock and SRS keep the law's record for each pulse; a Sine run names its law file by SHA-256 without copying it; the Random module records the name of its built-in law. The specification's aim, that every run keeps the law file and every call, is met by some modules and not yet by others.

4. The Drive Path

4.1 What It Is

The **drive path** is all code that can change what reaches an output channel, or whether and how a run stops (Cross and Zwink 2026, `docs/drive-path.md`). That includes more than the code that writes a drive: the limits a drive is judged against, the files those limits are read from, the loop's timing, the control laws, the control service and its protocol, and the thread a run's data is written from.

A file in the repository lists it by path, each with a reason, in two sections: drive path and not drive path. Every crate, module, law, interface and protocol folder must be on exactly one of the two lists, and the automated checks fail on a new crate until someone classifies it. The question to ask is whether the code can put anything on an output channel, change a limit or a ceiling a drive is judged by, or change when or how a run ends, directly or through code that does. If it can, it is drive path.

Not on it: the analysis service, the shell, the readers and writers of recorded data, and the Recorder module, which records inputs and opens no output.

4.2 The Invariants

Seven rules hold in every build, whoever changes what (`docs/drive-path.md`, "The Invariants"):

1. The drive ceiling and the drive abort hold at all times, in every module.
2. A stop always ramps down; an abort always takes the drive to zero at once (on the NI backend, through a 25 ms taper, Section 5.5).
3. Any failure on a run thread ends in the emergency stop. Failures are caught on the run thread and the emergency stop is issued from there.
4. Every protection lives in the control service. The shell may show a limit, but it is never the only thing between an operator and the drive.

5. Only the control service drives, and only through the run commands of the control protocol, which pass its arm checks. The analysis service has no route to a drive.
6. Anything read from a file that feeds a test (a reference, a test setup, a shaker record, a channel table or a control law) is checked like a value typed in, and the operator sees it before arming. A channel table restored from a file or kept across a restart is refused at arm until the operator applies it.
7. The loop thread does no unbounded work. File reading, the network and anything else that can block stay off it.

4.3 The Safety Tests and the Rules for a Change

A second file lists the tests that protect a shaker, one line each, naming the crate, the file and the test, and what it protects: the drive ceiling, the drive abort, the stop ramp, the abort to zero, a run-thread failure, the law guards, the envelope, the kinematic limits, divergence, overload, underrun, a dropped controller, a late loop, the input watchdog, the arm checks, checked inputs, who may drive, and the shutdown. The automated checks confirm that every listed test exists, is a test, is not marked to be skipped, and is not left out of the build on every platform the checks run on; then the tests run with the rest.

A pull request that touches the drive path must carry a test that fails before the change and passes after (or say why that cannot be done and what was checked instead), and a section answering four questions: what can now reach the drive, how it fails safe and the test that shows it, what ran on the simulator, and what ran on hardware. The checks fail until the section is there. A pull request that removes a safety test, takes a path off the drive-path list, or loosens a limit, threshold, timeout or default on it must say so in its title, `[weakens safety]`, and be approved by the control maintainer. The checks catch the first two; for the third they flag each listed test whose body the change edits, and the reviewer judges.

The guide that states these rules is written for "a person or a coding agent" who has not worked on the controller before. Much of Hertzforge is written by AI coding agents, and the rules are meant to be enforced by checks wherever they can be, not by a reviewer's memory.

Anyone may change the drive path. The rules do not restrict who works where. What protects a shaker is the rules a change meets, the checks that enforce them, and a run on a bench before a release.

5. The Hardware Abstraction: What a Backend Has Proven

Test modules do not know which hardware they run on. They ask a hardware abstraction, behind which each backend states its capabilities. This is what "hardware-agnostic" means in Hertzforge: a new kind of hardware is a new backend, and no test module changes. Two backends exist, the simulator and NI-DAQmx, so the claim has been exercised on two.

5.1 Who Closes the Loop

Eigen was designed from its start for two kinds of hardware: a controller that runs its own control loop, and plain data acquisition hardware on which the program closes the loop itself. So the abstraction records **who closes the loop** as a property of the backend, and the question "can this test run here?" depends on the pair of test module and backend, not on the backend alone. A classical shock pulse played once, a random test closed every loop, and a waveform corrected between plays need different things from the hardware.

Hertzforge has only backends on which the host closes the loop. The interface for the other kind is kept, unused, so such a backend could be added.

5.2 One Resolver, and No Silent Downgrades

A test module asks for what it needs: where the drive comes from, and who corrects it and when (every loop, between plays, or not at all). A single function answers each request with exactly the request, or a refusal with a reason the operator sees word for word (`resolve_loop` in `crates/hertzforge-module-abi`). It never returns a different answer than the one asked for. A test that asks to converge on a target, on hardware that cannot return the response, is refused; it is not quietly turned into open-loop playback. The operator reaches playback by asking for playback.

History. Eigen's design record of July 2026 specified this resolver, but an audit of Eigen's hardware independence on 16 July 2026 found it had never been built as specified: a stub ignored the module and was called from nowhere, and three substitutes had grown in its place. One was a capability table in the desktop application's own code that listed five native modes for a device whose backend declared one. When the controller was prepared for Hertzforge at the end of October 2026, the authors settled that whether a mode may run is checked in the backend, through the resolver, for every mode and backend, and never only in the user interface. The resolver's own documentation in Hertzforge names five crates that had each invented their own local vocabulary for the same question before it existed.

5.3 A Capability Is a Claim of Proof

A backend's list of what it can do is not a list of finished code. It is a claim that hardware has shown it. The order is fixed: a mode is tested on hardware, then the backend declares it, then the resolver permits it. Declaring a capability because the code is written would remove the only check between code that has never run on hardware and a real shaker.

5.4 The Clock Must Be Proven, Not Assumed

Closed-loop random control builds the drive as a continuous stream of overlapping frames, and closed-loop sine control tracks a tone through a narrow filter. Both need the drive and the measurement to run on the same physical clock. On two clocks, a small rate difference accumulates into a phase drift that corrupts the frame overlap and detunes the tracking filter over a sweep. Eigen's design record therefore had the controller refuse to arm these modes on any backend that could not show one shared clock.

The NI hardware made the point concrete. The NI 9234 input module is a delta-sigma converter with its own oversampling clock; the NI 9263 output module is clocked on a different timebase. Starting both from one trigger is not the same as running them on one clock.

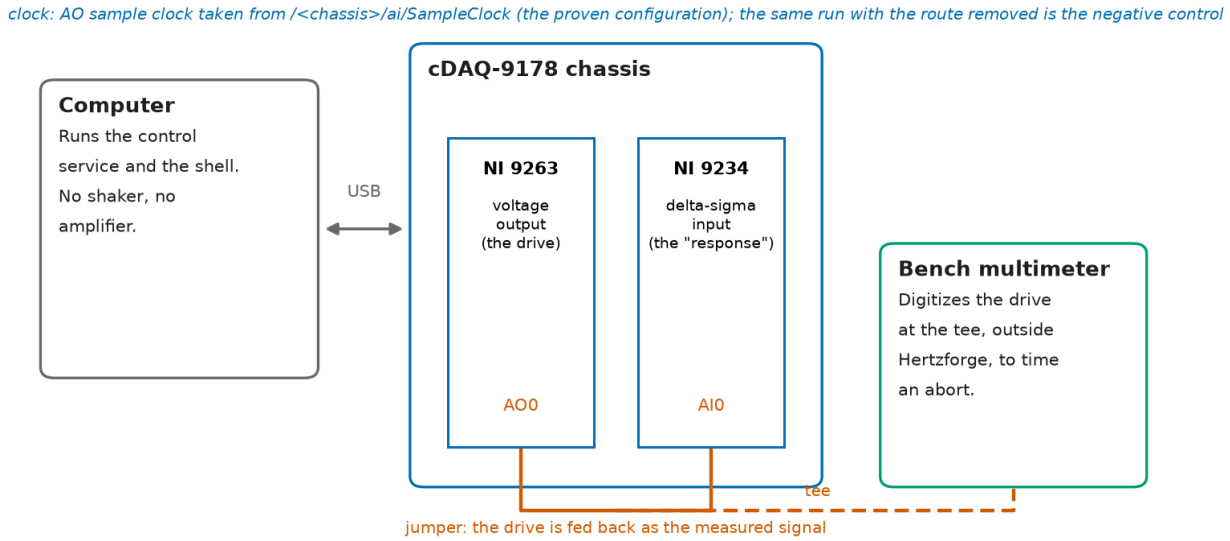
So each backend reports its **clock domain** as shared, independent or unknown, and unknown is the default. The NI backend reports shared in only two cases (`crates/hertzforge-hal-ni/src/clock.rs`):

- **The driver reports every device as simulated.** A simulated device has no physical clock and no shaker can be attached to it; every run made this way carries a SIMULATED label. There is no flag, setting or environment variable that turns this on for real hardware; the only input is the driver's own answer.
- **The exact configuration is in a proven list:** the chassis, the input module, the output module and the clock routing, each entry carrying the bench capture that proved it. A different module pair or chassis must prove it again.

The list has one entry: a cDAQ-9178 chassis with an NI 9234 and an NI 9263, the output's sample clock taken from the input's (Figure 3). It was proven on 30 July 2026 with the output wired to the input, a 1 kHz tone played for 300 seconds, and the accumulated phase measured. With the clock routed, the drift was +0.0051 parts per million and flat. With the route removed, as a negative control on the same bench, it was −322.6 parts

per million and accumulating, which shows the measurement can see unlocked clocks. A second tone at 700 Hz agreed.

A correction made between plays (time waveform replication, SRS iteration) does not need a shared clock, because each play stands alone. Those modes are not held behind the clock proof; Random and Sine are.



5.5 The Abort Must Be Fast Enough, and Measured

Each backend also states a bound on how long an abort takes to act. A hardware station that would drive and states no bound refuses to arm. For the NI backend the bound is derived from the input module's published filter delay (NI 2026) and the output blocks already queued, and it came to 160.75 ms at 8533.33 Hz.

On 1 August 2026 the bound was tested on the bench of Figure 3, with the multimeter digitizing the drive outside the program, since the program's own input stops with the drive. From the moment the input crossed its limit, the drive stopped rising after 50.3 ms (the output already queued), then ramped to zero in 23.1 ms, 73.4 ms in all, ending at 0.9 mV. The derived bound was conservative, which is the right direction for a safety bound. The same test found that, before it, the drive did not go to zero on an abort; it froze at the level it had when the limit tripped. The fix tapers the drive to zero over 25 ms. A derived bound did not show that; a measurement outside the program did.

5.6 A Defect That Shaped the Rules

The same Eigen audit of July 2026 found a defect in the loop's scheduler. The backend reported how many frames it had read (one frame is one sample on every channel), and the scheduler took that number as a count of samples and divided it by the number of channels. With one channel the two are the same. With N channels, the scheduler processed only the first 1/N of every block and counted it, consistently, as 1/N of the frames, so nothing looked wrong. The abort watchdog saw the same fraction. In a test written to prove it, a 99 g excursion against a 10 g limit did not trip. Every test until then had used one channel.

The fix made the unit part of the type: the read now returns `Frames(n)`, and samples are derived from frames. The defect also changed how the authors read code. A test double in another module had written the wrong unit into its documentation as the contract, and of ten defects found in those two days, each had been hidden by the code's own documentation, not by the lack of it. When a comment and the code disagree, the code is taken as the truth and the comment is removed until it is checked.

6. The Simulator Is a Backend, and What It Hides

6.1 One Code Path

The simulator is a backend like the NI one. It holds a simulated structure (modal models, with level dependence where a study needs it) and answers the same calls. The same test module code, the same control law and the same protections run against it as against hardware. A run on the simulator is always marked SIM and is never presented as a hardware result; a reader that finds no label refuses or marks the source unknown, and never assumes one.

This is what lets control laws be developed and compared on simulated structures, and lets a whole test, through the real control service and the shell, be run in an automated end-to-end check with no hardware attached.

6.2 What the Bench Found That the Simulator Did Not

The simulator reads a new drive on the next sample. Hardware does not. On the NI bench, the backend keeps the output queue two blocks ahead (512 frames at 12,800 Hz), and the system identification measures a delay of 553 frames (43.2 ms) where the simulator measures none.

That difference mattered. During the conversion of the Sine module in November 2026, the built-in Sine law's default compression (how fast the amplitude servo responds) had been changed to "fast" on simulator evidence. On the loopback bench, with the law holding a dwell through +6 dB level steps, fast overshot each step by 4.3 and 4.5 dB at 20 Hz and by 1.2 and 1.6 dB at 100 Hz, and was still ringing 2.4 seconds later at 20 Hz. Slow stayed within 0.04 dB. The default was set back to slow (Cross and Zwink 2026, commit `75b2270`). The same bench found a template law that held for a fixed number of loops after starting and then corrected an error it had not caused, because the measured response had not yet arrived; it now waits until the measurement reads what the identification predicts (commit `fa80084`). The template had passed the law check, and the built-in's default had been tuned on the simulator; neither the check's simulated stations nor the simulator had any delay.

Other modules' bench runs added more of the same kind. The output module holds each sample until the next, with no reconstruction filter, so a measured response must model the hold. A station asked for one sample rate had been configured at the device's default rate instead. And in the Recorder, the thread that reads the hardware held a lock across each read; on the simulator, which sleeps between batches without the lock, nothing showed, but on hardware the read is the pacing, and commands went unanswered for 5 seconds and an abort took about 10 seconds to end acquisition, until the reader was made to give way to a waiting command (commit `df06010`).

The lesson taken from these: a control choice measured only on the simulator is provisional. Since then the authors' practice has been to run each closed-loop module's default law on the loopback bench, and measure its start and its response to a level step, before it is merged. It is a practice, not yet a written rule of the repository.

7. A Run Is Kept for Analysis

A test is worth what it leaves behind. Every control module (Random, MIMO Random, Sine, Mixed Mode, Classical Shock, SRS, MIMO Shock, TWR, Explorer and the Recorder) writes a **run folder** while it runs (Cross and Zwink 2026, `docs/run-folder.md`):

```
<run name>/
  run.json           the manifest
  table.json         the channel table the run armed with
  setup.json         the configuration as accepted, and what it resolved to
  law/law.json       the control law in use
```

```
log.jsonl           the run's log, one entry per line
streams/<name>/     telemetry.f32 and stream.json
results/            what the module computed
end.json            written once, at the end
```

The manifest names the source, SIM or HARDWARE, from the station's class, and is never written without it. It names the station's devices with their serial numbers and clock domain, the sample rate asked for and the rate the hardware actually programmed, and the build of the program that ran. A run folder is never written over: the code claims a new folder with a call that fails if the folder exists. All file work happens on a writer thread; the loop thread hands data over without waiting, and a failed write during a control run is a notice, never a stall or an abort. The Recorder differs in one way: recording is its purpose, so a failed write ends a Recorder run.

The analysis service converts a run folder into an ordinary project, so a finished run reaches every analysis tool with its channel table, setup, log and measurements, without anyone retyping them. A project is a `.hforge` file: a standard HDF5 file (The HDF Group 2026) with a documented layout, which started as Visual Dynamics' project schema. Renamed to `.h5`, it opens in general HDF5 tools, and the layout is written down so a file can be read without Hertzforge.

Two rules keep the two sides consistent. A conversion between a control-side form and an analysis-side form is written once, in Python; a copy in another language exists only as a test against shared sample data. And every control module's acceptance list includes "the finished run opens on the analysis side with its channel table and setup", recorded as not met until it is.

History. Eigen chose HDF5 for capture from the first sample in July 2026, for files that stay readable after a crash and because two widely used scientific formats are themselves HDF5. Hertzforge kept HDF5 for the project file, but records a run as plain streams during the run and converts it after, so a run that is killed partway leaves a readable folder.

8. What Carried Over From Eigen, and What Changed

Table 2: Eigen's main design decisions, July to October 2026, and what became of each in Hertzforge.

In Eigen	In Hertzforge
A control core in Rust, kept free of any interface framework	Kept. It is the control service
An interface in TypeScript and Svelte, in a Tauri window	Kept, and grown into the one shell for control and analysis
A backend process that owns the hardware, with the interface as a client over a versioned protocol	Kept, and joined by the analysis service on the same protocol
Each test module chooses its own control strategy; no single global loop	Kept
Who closes the loop resolved for each pair of module and backend	Kept; the resolver was built
Closed-loop modes refused until a shared clock is proven on a bench	Kept, with a proven list keyed to the exact configuration
Capture in HDF5	Kept for the project file; runs recorded as streams and converted after

In Eigen	In Hertzforge
Closed source, sold by module	Changed: MIT license. A law file is a separate work whose author chooses its license; the laws shipped with Hertzforge are MIT
Modules as signed native libraries, also the license boundary	Changed: control laws as sandboxed WebAssembly files
Offline license files	Dropped
A backend for a controller that runs its own loop	Not carried into Hertzforge; the interface for one is kept
Pass criteria for control laws written for a commercial product	Loosened for laws: a law useful on most tests is offered with its limits stated. Not loosened for protections

The last row is the main change in outlook. The pass criteria carried over from Eigen were written for a commercial product: tight tolerances at every control, and research plans that required every simulated case to pass. As an open research platform, Hertzforge offers laws that work on most tests, with the cases they fail stated, because a user can choose another law or write one. That is safe to do only because the protections in Sections 3.3 and 4 do not depend on which law runs.

9. Limits

- **Loopback evidence.** The hardware measurements quoted here come from loopback runs, with the output wired back to the input (Figure 3).
- **Two backends.** The hardware abstraction has been exercised by the simulator and by NI-DAQmx. The claim that new hardware needs only a new backend rests on two cases.
- **One proven clock configuration.** Closed-loop Random and Sine arm on real NI hardware only on the one chassis and module pair in the proven list.
- **One measured abort.** The abort time of Section 5.5 was measured on one configuration at one sample rate.
- **The law interface is version 0.x** and may still change.
- **Some rules rest on review.** The checks enforce the drive-path lists, the presence of the safety tests and the pull request sections. Whether a change loosens a limit, and whether a test is adequate, are judged by a reviewer.
- **The simulator's known gaps** (Section 6.2) are not all modeled in the law check. A law can pass the check and still behave differently on hardware.
- **This describes one commit.** Hertzforge changes daily. The principles in this article are meant to last; the file names and lists will not, and the repository at a later commit is the authority on them.

Contribution Statement

- **Kevin Cross:** Conceptualization, Software, Supervision, Writing: review and editing.
- **Brandon Zwink:** Conceptualization, Software, Writing: review and editing.

How This Was Made

AI tools were used in this work. Much of Eigen and Hertzforge was written, and this article drafted, with Claude Code, an AI coding assistant running Anthropic's Claude Opus models, under the authors' direction. The authors are responsible for every word and number.

Acknowledgments

The idea of control laws a user can load into an open vibration controller, and much of the input set for a MIMO random law, comes from Rattlesnake, the open-source controller of Sandia National Laboratories, as described in its published paper and user's manual; credit for those concepts goes to Sandia National Laboratories and the Rattlesnake authors.

References

- Bytecode Alliance (2026a). The WebAssembly Component Model. https://component-model.bytecodealliance.org/
- Bytecode Alliance (2026b). Wasmtime. https://wasmtime.dev/
- Cross, K., Zwink, B. (2026). Hertzforge, main branch at commit b587d5a1. https://github.com/hertzforge/hertzforge
- NI (2026). NI 9234 Specifications. https://www.ni.com/docs/en-US/bundle/ni-9234-specs/page/specs.html
- Project Jupyter (2026). JupyterLab. https://jupyter.org/
- Rohe, D., Schultz, R., Hunter, N. (2021). Rattlesnake: An Open-Source Multi-Axis and Combined Environments Vibration Controller. IMAC XL, Orlando, 2022. SAND2021-13012C, Sandia National Laboratories. https://www.osti.gov/servlets/purl/1893280
- Sandia National Laboratories (2026). Rattlesnake Vibration Controller User's Manual, section 12.7, custom control laws for MIMO random vibration. https://sandialabs.github.io/rattlesnake-vibration-controller/book/print.html
- The HDF Group (2026). HDF5. https://www.hdfgroup.org/solutions/hdf5/
- Zwink, B. (2026). Visual Dynamics. https://visualdynamics.org/

Implemented In

Hertzforge, main branch at commit b587d5a1

Version History

Version 1, 2026-11-15. DOI 10.5281/zenodo.24260838. First published.

Reviews. This article is reviewed in public. Its reviews, its review status and any later version are at https://hertzforge.com/journal/2026-006/

Copyright 2026 Kevin Cross and Brandon Zwink. Text, figures and data CC BY 4.0; code MIT.

Methods are published for research. Anyone who applies one to a test is responsible for protecting the equipment and the article under test.