

Chronoc

Reference Manual

Version 1.0 · October 27, 2026 · Layer1Labs Silicon Inc.

The full reference for the Chrono target and the chronoc compiler: how a Lingua Franca program becomes a constraint specification, how engines execute it, and how to bind it in software or as soft IP.

Language baseline. This manual describes the Chrono target as built on **Lingua Franca v0.13.0**. Upstream Lingua Franca publishes no separately versioned language specification; the language is pinned by that release tag and the Lingua Franca handbook as of that release. The compiler build that provides the Chrono target reports itself as `lfc 0.14.0-SNAPSHOT` because it sits on the post-0.13.0 development line — the language baseline remains v0.13.0, and this manual says "v0.13.0" wherever the language itself is meant (Chapter 13 records every pin).

See it live: The documentation library — `https://layer1labs.ai/docs/`
This manual is published in the Layer1Labs documentation library: `https://layer1labs.ai/docs/`

© 2026 Layer1Labs Silicon Inc. All rights reserved. See the Notices at the end of this document.

Contents

How to read this manual **4**

1 The system in one chapter **4**

2 Architecture, end to end **5**

3 Why Chrono is not ordinary Lingua Franca **6**

4 The language subset: what a Chrono program may contain **7**
4.1 Program shape . . . 7
4.2 The driver reaction . . . 8
4.3 Reaction bodies . . . 8
4.4 Guards . . . 8
4.5 Preambles . . . 9
4.6 Target blocks . . . 9
4.7 State . . . 9
4.8 What is excluded, and why . . . 9
4.9 A complete program . . . 10

5 From program to schedule **10**

6 The compiled specification (.cspec, format CSP1) **11**

7 Intrinsics and the effector manifest **12**
7.1 What an intrinsic is . . . 12
7.2 The built-in table . . . 13
7.3 Writing a manifest . . . 13
7.4 Profiles: who owns an intrinsic name . . . 14

8 Execution: the kernel contract **14**
8.1 Load time . . . 14
8.2 The plan round . . . 15
8.3 The effector contract . . . 15
8.4 Refusals and retries . . . 16
8.5 The run report . . . 16
8.6 Edge cases, stated once . . . 16

9 Extensibility: imports, packages, and libraries **17**

10 Integration: the software engine (ChronoHive) **17**

11 Integration: the hardware tier (ChronoFabric) **18**

12 Conformance and portability **19**

13 Pins, versions, and provenance **20**

© 2026 Layer1Labs Silicon Inc.

14 Worked example: TrafficGrid **21**
14.1 The program . . . 22
14.2 The intrinsics . . . 22
14.3 The compiled specification . . . 22
14.4 Reading the example back through the manual . . . 23
14.5 The example in service . . . 23

A Manifest quick reference **23**

B Error and refusal behavior **24**

C Glossary **24**

D Companion documents **25**

Version history **26**

Notices **27**

© 2026 Layer1Labs Silicon Inc.

How to read this manual

Three reading paths, depending on what you are doing:

- **You write programs.** Chapters 1, 3, 4, 5, and 7, then the worked example in Chapter 14. Chapter 12 collects the rules that keep a program portable.
- **You integrate an engine.** Chapters 2, 6, 8, and 10 (software) or 11 (hardware), with Chapter 7 for the intrinsic contract your binding must honor and Chapter 12 for conformance.
- **You evaluate the system.** Chapters 1, 2, and 3 state what the system is and why it is shaped the way it is; Chapter 13 records the exact versions everything here is pinned to.

Conventions: a term is defined once, at its first use, and used in that sense throughout (Appendix C collects the definitions). Behavior is stated before mechanism, and failure behavior is stated explicitly wherever it exists — refusals are never implied. Where a rule or mechanism is adopted but still landing in the toolchain, the text says so at the rule. Nothing in this manual describes a planned capability as a present one.

1 The system in one chapter

Layer1Labs builds deterministic execution for AI and data infrastructure. The problem it answers is simple to state: the work that keeps expensive computation alive — checkpoint writes that protect progress, data movement that feeds the next step, the coordination of many small operations against shared, limited resources — is normally scheduled by heuristics that guess. When the guess is wrong, accelerators stall, and the stall is different every time, so it cannot be reproduced, audited, or reasoned about after the fact.

The Chrono system replaces the guess with a compiled artifact:

1. A designer writes a **program** in Lingua Franca declaring `target Chrono`. The program describes components, how they are wired, and the operations each component proposes — nothing more.
2. The **chronoc** compiler validates the program and compiles it into a single **constraint specification** (a `.cspec` file, format CSP1): the program's operations, their resource demands, their dependencies, the capacities they run against, and the full schedule, worked out ahead of time.
3. An **engine** executes the specification. At each step it decides — by arithmetic over the declared demands and capacities, not by heuristics — which proposed operations are admitted, runs the admitted work through **effectors** (the real functions, or hardware endpoints, that do the work), observes completion, and records what happened in a run report.

Two engines execute the same specification: **ChronoHive**, the software engine, binds effectors to functions in a host process; **ChronoFabric**, the hardware tier, binds the same effector names to endpoints in soft IP on FPGA platforms. The specification does not change between them; only the binding differs. That sentence is the product claim, and this manual is written so a reader can check it: Chapter 6 documents exactly what the specification contains and pointedly does not contain, Chapter 7 documents the contract that makes an effector name mean the same thing on both tiers, and Chapter 12 states the rules an engine must satisfy before it may claim to run a program.

Determinism is the point of the design. A run's decisions depend only on the specification: the same specification admitted to the same capacities produces the same admissions, the same order, and the

© 2026 Layer1Labs Silicon Inc.

same refusals on every run. Time inside a run is a synthetic clock derived from the schedule, never the wall clock, so nothing in a decision can drift with machine load. What an operator gets back — the run report — is therefore a record that can be replayed and audited, not a log of one unrepeatable afternoon.

2 Architecture, end to end

The pipeline has five stages. Each stage's output is the next stage's complete input — no stage reaches back around another.

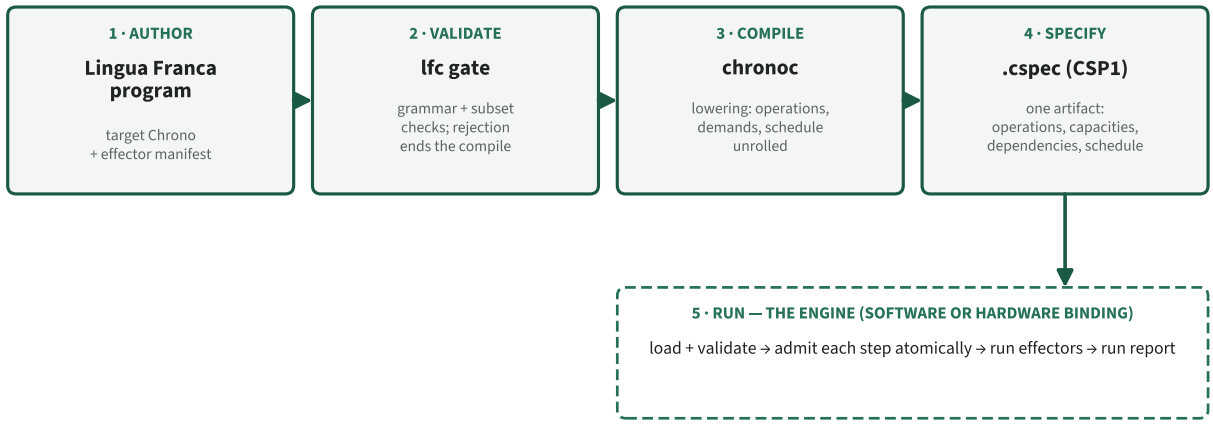


Figure 1: The pipeline: program and manifest in; gate, compiler, specification; the engine runs the identical artifact on either binding tier.

The program and the manifest. The inputs are a Lingua Franca source file declaring `target Chrono` (or `target Python` for existing programs — the specification is identical either way) and an effector manifest declaring the intrinsics the program may call (Chapter 7). The program states *what work is proposed and when*; the manifest states *what each proposed operation is called at run time, what it demands, and what happens if it is refused*.

The validation gate. Before chronoc lowers anything, the pinned Lingua Franca compiler validates the program against the declared target's grammar. This gate is mandatory: a program the gate rejects is a compile error, and no specification is produced. chronoc then validates the program against the Chrono language subset (Chapter 4) — a narrower set of rules than the full language, each rejection naming the construct that broke it.

Lowering. chronoc evaluates the program ahead of time: it resolves parameters, walks the reactor instances and their wiring, and unrolls the program's timers and counters into a fixed schedule of steps (Chapter 5). Every intrinsic call becomes one operation template carrying its effector name, its computed demands, its refusal policy, and its dependencies. The result is written as one `.cspec` file (Chapter 6).

Load and validation. An engine reads the specification — never the source. The reader validates the file completely (structure, checksum, bounds, that every demand names a budgeted resource, that dependencies form no cycle) before anything is constructed. The host then supplies an effector for every effector name in the specification; if any is missing, the run is refused before anything executes, and the refusal names what is missing. Under the intrinsic-profiles model (Chapter 7), loading is also where the engine's published capabilities are checked against the specification's recorded requirements — a mismatch is refused at load, not discovered mid-run.

© 2026 Layer1Labs Silicon Inc.

Execution. The kernel runs the schedule one step at a time. Each step is one *plan*: the set of operations proposed for that step, plus any carried retries from earlier steps. The kernel admits the plan atomically — it checks the whole step's combined demands against the capacities before any of it is committed — then the admitted operations run in dependency order, each through its bound effector, and a dependent operation starts only after the completion of the operations it depends on has been observed. Refusals are not errors in themselves; each operation carries a policy that says what a refusal means (Chapter 8).

The run report. The run returns a record: which operations were admitted, refused, retried, and dropped, per operation; the steps completed against the schedule's length; the capacities in force; and the provenance of the specification that ran. Because execution is deterministic, the report is a property of the specification, not of the machine that ran it.

Two properties of this architecture are worth naming before the detail chapters. First, **the engine does no I/O of its own**: it decides whether and when work may run; effectors do the work. That separation is what lets the same specification bind to functions in a process on one tier and to hardware endpoints on the other. Second, **admission happens before execution, never during it**: by the time an effector is called, the decision that it may run — and that its resource demands fit — has already been made, once, for the whole step.

3 Why Chrono is not ordinary Lingua Franca

This chapter is the design statement of the system. It is worth reading even if you skip the reference chapters, because it is the claim boundary everything else in this manual is written inside.

In ordinary Lingua Franca, a target names the language the compiler generates code in: write a program, choose `target C` or `target Python`, and the compiler emits code in that language that *is* the program's implementation. The program text and the running software are the same thing, one step removed.

The Chrono target breaks that pattern deliberately:

- **No target-language code is generated or run.** A Chrono program compiles to a constraint specification — data, not code. The specification contains operations, demands, dependencies, capacities, and a schedule. It contains no instructions for any processor, in any language.
- **The program is a schedule and a contract, not an implementation.** A reaction body in a Chrono program may not contain arbitrary code. It may call *intrinsics* — named operations the compiler recognizes — set output ports, and maintain the counter that drives the schedule. What an intrinsic *does* is never written in the program. It is declared in the effector manifest (what it is called at run time, what it demands, what a refusal means) and implemented behind the engine's binding, by whoever provides the engine.
- **The same artifact runs on engines that share nothing but the specification.** Because the program carries no implementation, nothing in it prefers one engine. The software engine binds an effector name to a function; the hardware tier binds the same name to an endpoint in soft IP. The specification is the whole agreement between the program and either engine.

What this buys:

- **Auditability.** The complete behavior a program can exhibit — every operation it can propose, every demand it can make, every step it can run — is finite, declared, and inspectable in one artifact before anything executes.

- **Admission before execution.** Because demands are computed at compile time, an engine can decide whether a step's work fits its resources before running any of it, and refuse honestly when it does not. A heuristic scheduler cannot offer this; it finds out by failing.
- **Binding portability.** A program is portable in the strong sense: not "the source recompiles elsewhere," but "the identical artifact executes elsewhere, with the same decisions," provided the destination engine honors the intrinsic contract of Chapter 7.

What it costs, stated as plainly:

- **A deliberately small language.** The Chrono subset (Chapter 4) accepts exactly the programs whose demands and schedules the compiler can work out completely, ahead of time. Constructs that would make demands uncomputable — loops with data-dependent bounds, arbitrary functions, physical-time behavior — are rejected at compile time, with the construct named.
- **Semantics must be specified, not just written.** In ordinary LF, an intrinsic's meaning would be its code. Here, code is exactly what the program does not carry — so the meaning of each intrinsic has to exist as a written, testable contract (Chapter 7), or portability is a promise nobody can check.
- **Some constants live outside the language.** The subset states integers, not floating-point values. A parameter an effector needs as a float — a gain, a clamp — cannot be written in the program; it belongs to the effector's implementation, and Chapter 7's discipline makes such implementation-owned constants part of the contract an engine is checked against, rather than invisible details.

One sentence in this chapter deserves emphasis because summaries of this system get it wrong: **no planner arithmetic is claimed to be written in Lingua Franca.** The program declares the schedule, the operations, and their demands. The arithmetic inside an effector — how a traffic plan is computed, how a training step is scheduled in detail — is the effector provider's implementation, bound by name, checked by the conformance rules of Chapter 12, and deliberately not described in this manual.

4 The language subset: what a Chrono program may contain

The subset is defined by what chronoc must be able to do with the program: evaluate it ahead of time, completely. Every rule below exists because some part of the schedule, the demands, or the dependencies must be computable at compile time. A program that breaks a rule is rejected at compile time; the rejection names the construct and, where the subset's error texts do so, the acceptable alternative. Rejection classes and their compile-time behavior are collected in Appendix B.

4.1 Program shape

A Chrono program consists of exactly one **main reactor**, in a file whose name is the main reactor's name with the `.lf` extension. The main reactor is the whole program: there is no second top-level structure. A main reactor's body may contain five kinds of member, and nothing else:

- **state variables** (`int` only in this version; see Section 4.7),
- **timers** (`timer t(step, period)` — period unit-less, in steps),
- **ports** (`input` and `output`; port types are not enforced and are best left untyped),
- **reactions**,

- **child reactor instances**, with the connections (`->`) that wire them together.

A reaction takes at most one trigger, and the trigger is a timer. A program's time is therefore entirely its timers: there are no physical actions, no external-event triggers, and no shutdown or startup reactions in the pipeline. Each step of the schedule corresponds to one timer firing (Chapter 5).

4.2 The driver reaction

Every Chrono program has exactly one **driver reaction**: a reaction triggered by the main reactor's timer whose body increments a counter and stops the program when the counter reaches a declared bound:

```
reaction(t) {= # Python target code
    c = get_state(c) + 1
    # ... intrinsic calls and port sets ...
    if c >= steps:
        request_stop()
=}
```

The bound (`steps` here) is a parameter or a constant, and must be an integer between 1 and 1,000,000 inclusive. The bound is the program's declared horizon: chronoc unrolls the schedule to exactly that many steps (Chapter 5), so a program cannot run longer than it declared, and an engine never has to guess when a run ends. A program without a well-formed driver reaction has no determinate length and is rejected at compile time.

4.3 Reaction bodies

A reaction body may contain only:

- **intrinsic calls** — calls to names declared in the effector manifest (Chapter 7). An intrinsic call is the program proposing an operation.
- The **counter increment** (exactly one per driver reaction; see Sections 4.2 and 4.7 for how the counter is kept).
- **Port sets** (`port.set(value)`) — how a reaction hands a value to connected reactions in the same reaction round.
- **`request_stop()`** inside the driver's stop guard.

No arbitrary code is permitted in a body: no other calls, no computed expressions beyond the two guard shapes below, no loops, no imports of host-language libraries. This is the load-bearing rule of the subset — it is what makes every operation and every demand in the program a compile-time fact.

4.4 Guards

A body may test its counter with exactly two conditional shapes, and no others:

```
if <counter> % <k> == 0:      # periodic: every k-th step
if <counter> >= <param>:     # horizon: at/after a declared bound
```

<k> is an integer constant; <param> is a program parameter (so a caller can set the threshold per instance). There is no `else`, and no other operator is admitted in a guard. Between them the two shapes express the scheduling patterns deterministic systems actually use: unconditional work, periodically gated work, and work that begins (or ends) at a declared step.

4.5 Preambles

A program file may carry a preamble (host-language code the generated runtime would share). Two facts about preambles matter:

- Preamble code is **never executed** in a Chrono run — there is no generated runtime. Preambles in the Chrono subset serve one purpose: they declare **imports** of other Lingua Franca files (Chapter 9).
- Because imported reactors are resolved by the pinned LF compiler's pre-generation pass, preamble imports must appear before validation; chronoc itself parses one file. The import story, and its current limits, is Chapter 9.

4.6 Target blocks

A program declares `target Chrono` or `target Python` (both compile to the identical specification; "Chrono" is the name used throughout this manual). The subset reads these target properties: `timeout` (parsed; not used by the pipeline), `workers` (must be `0` — greater than zero is rejected, because parallel host workers are incompatible with the deterministic-ordering guarantee), and `trace` (parsed; no effect in this version). Target properties outside this set are rejected at compile time.

4.7 State

State is deliberately minimal. A state variable is declared `state c:int = 0`, and a body reads it with `get_state(c)`. The counter pattern — one integer counter, incremented once per driver firing, tested by the two guard shapes — is the only state pattern the subset defines semantics for. There is no `set_state` in the pipeline: the driver counter is the program's clock made visible, and programs that need other state keep it inside their effectors, where it belongs to the implementation, not the schedule. (A program source may contain other forms for other tools; chronoc defines behavior only for the counter pattern and rejects programs whose state use it cannot evaluate ahead of time.)

4.8 What is excluded, and why

Excluded	Why (each keeps demands or the schedule uncomputable ahead of time)
Physical actions, startup/shutdown reactions	Run length and step timing would depend on the host, not the program.
Multiple triggers per reaction; port-triggered reactions in the driver	The schedule must be a function of the timers alone.
Data-dependent loops or branches beyond the two guards	An operation must exist (or not) as a compile-time fact per step.
Floating-point parameters	Demands are integers; floats belong to effector implementations.
`workers > 0`	Parallel host execution breaks the deterministic order guarantee.
Arbitrary host code in bodies	The manual's core boundary (Chapter 3): bodies declare work; they do not perform it.

4.9 A complete program

This program is small, valid, and used again in Chapter 5 to show unrolling. It proposes a training operation every step, a checkpoint on every third step, and a prefetch whenever the operation reactor reports a step done:

```
target Chrono {
    workers: 0
}

main reactor Trainer(steps:int = 6, ckpt_every:int = 3) {
    timer t(0, 1)
    state c:int = 0
    output done:bool

    reaction(t) {=
        c = get_state(c) + 1
        train_step()
        if c % ckpt_every == 0:
            admit_checkpoint(mb=512)
        prefetch(depth=2)
        done.set(True)
        if c >= steps:
            request_stop()
    =}
}
```

5 From program to schedule

chronoc's core job is turning a program — which talks about timers, counters, and guards — into a schedule — which talks about steps and the operations each step proposes. The transformation is mechanical, and this chapter states it exactly.

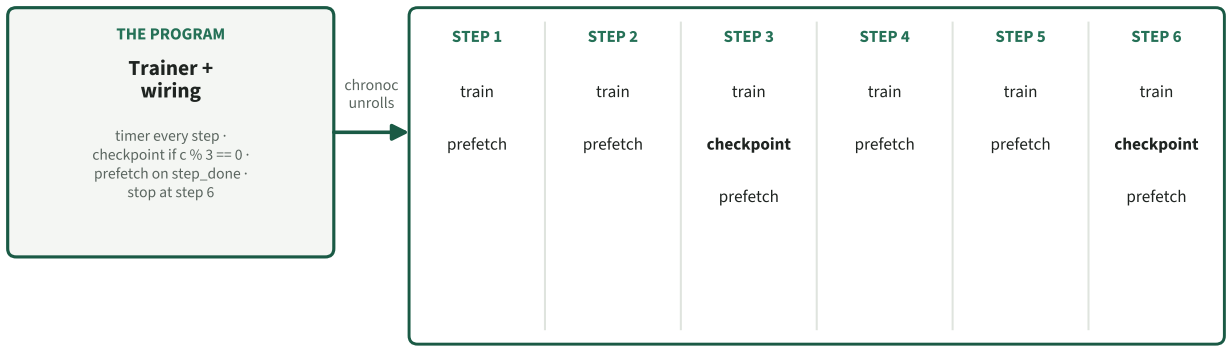


Figure 2: Unrolling the Section 4.9 program at its declared defaults: the guard `c % 3 == 0` admits a checkpoint into steps 3 and 6 only.

Parameters resolve first. Constants, parameters, and their defaults are evaluated; every quantity the unrolling needs — step counts, guard moduli and thresholds, call-site argument values — must reduce to a compile-time integer. A quantity that cannot be so reduced is a compile error, not a run-time surprise.

Instances are flattened. The reactor tree (main reactor, its children, theirs) is walked and flattened into instances, each with a stable identity under its **instance path** (for example, `trainer`, or `grid.cell3`

for a child named `cell3` of an instance named `grid`). Instance paths are how operations name where they came from: every operation in the specification carries the path of the instance whose reaction proposed it (the *binding*), and every dependency traces back to a connection or port set between instances (see the next paragraph).

The driver defines the steps. The driver's stop bound `N` (1 to 1,000,000) fixes the schedule's length: steps `1..N`. The timer period is checked against the step definition — in this version every driver timer fires every step, and the specification's `step_ns` (the duration one step names, in nanoseconds) is recorded in the specification header for consumers that report time.

Guards are evaluated per step. The counter starts at its declared initial value and increments once per step, exactly as the driver body does. For each step, each guard is evaluated against that step's counter value: a guarded intrinsic call becomes an operation in the steps where its guard is true, and in no others. Unguarded calls appear in every step. The result is that *which operations exist in which steps* — the entire question the schedule answers — is settled completely at compile time.

Dependencies come from the wiring. Within a step, operations are ordered by the program's connections: if instance A's output feeds instance B's input and the corresponding operations run in the same step, A's operation precedes B's. A reaction that sets a port is likewise ordered before the reactions its value reaches. The per-step order written to the specification is topological: every operation appears after all operations it depends on. A cycle in the wiring is a compile error — there is no order that satisfies it, and the compiler says so rather than picking one.

What the engine receives. For each step: the list of operations proposed in that step, each with its template identity (template IDs are assigned in encounter order — the first distinct call site encountered is template 0, and so on), its effector name, its refusal policy, its computed demands, and its dependencies. Nothing about how the list was derived is needed at run time; the derivation is this chapter, and the artifact is Chapter 6.

6 The compiled specification (.cspec, format CSP1)

The specification is the contract between compilation and execution. This chapter describes the CSP1 binary format at the level an implementer of a reader — or a skeptic — needs: its sections, its invariants, and what it refuses to contain.

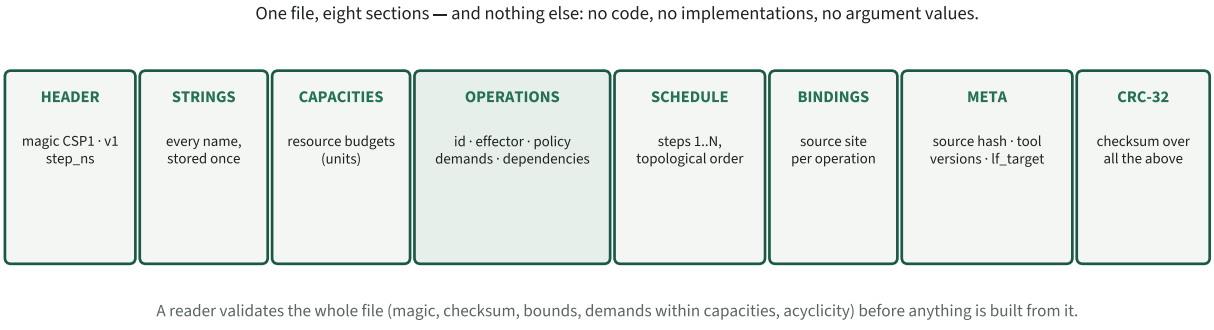


Figure 3: The eight sections of a CSP1 file, in order. A reader that encounters anything else — or any section out of order — rejects the file.

Header (fixed size). Magic bytes `CSP1`, a format version (1), flags (0 in this version), and `step_ns`, the duration one schedule step names, in nanoseconds.

String table. Every name used anywhere in the file — resource names, effector names, instance paths, version strings — stored once, as UTF-8, and referenced by index thereafter. Names in a specification are data, never code.

Capacities. The resource budgets the program's demands run against: for each resource, a name and a capacity in integer units. Capacities travel with the specification so that admission means the same thing to every reader; Chapter 8 states how an engine may lower (never raise) them at load, and how the run report records the capacities actually in force.

Operations. One record per operation template: a template ID; the effector name (index into the string table); the refusal policy (`must-admit`, `retry`, or `drop`); the dependencies (template IDs that must complete earlier in the same step); and the demands — for each, a resource name and an integer amount in that resource's units. Demand amounts are the manifest's computed values (Chapter 7): by the time an operation exists, its demand is a number, not a formula.

Schedule. For each of the N steps, the operation IDs proposed in that step, in topological order (Chapter 5).

Bindings. For each operation, the instance path of the reactor instance that proposed it (Chapter 5) — the provenance that lets a run report say *where* work came from, and lets the hardware binding route an operation to its endpoint (Chapter 11).

Meta. Provenance of the artifact itself: the SHA-256 of the Lingua Franca source file; the `lfc` version string of the compiler that produced it; the chronoc version; the LF target the program declared; and `target_steps`, the declared horizon. Two notes. The source hash covers the main source file; programs assembled from imports carry the limitation discussed in Chapter 9. And the versions recorded here are what the producing tools reported about themselves, which is why Chapter 13 pins their exact forms.

Checksum. A CRC-32 over all preceding bytes. A file whose checksum does not match is rejected whole.

Reader validation is total. A conforming reader checks, before it builds anything: the magic and version; the checksum; that every index and offset is in bounds; that every demand names a resource with a stated capacity and does not exceed it singly; that dependencies are acyclic and within the same step. Any failure rejects the file with a named error; there is no partial load, and no repair. Appendix B lists the validation failures and their meanings.

What the format does not contain is as normative as what it does: no code of any target language; no effector implementations; no argument values beyond the integers the compiler folded into demands and schedule decisions (an intrinsic's argument influences the specification only through the demand it computed); no wall-clock times; no host-specific data. An artifact obeying this chapter can be produced by the compiler of Chapter 2 and consumed, unchanged, by both engines of Chapters 10 and 11 — which is the property Chapter 3 promised.

7 Intrinsics and the effector manifest

An **intrinsic** is the seam of the whole system: the name a program calls, the declaration that gives the name a runtime meaning, and the contract that lets two engines honor the same name identically. This chapter defines all three parts.

7.1 What an intrinsic is

A program proposes work by calling an intrinsic — a name such as `train_step`. The name alone means nothing. The **effector manifest** is the declaration that completes it: for each intrinsic, the manifest states the **effector** it lowers to (the runtime name the engine binds), the **refusal policy** (what a refusal means for that operation), the **arguments** the call takes, and the **demands** it computes. chronoc refuses to compile

a program that calls a name no manifest declares: an undeclared intrinsic is a compile error, because an operation with no declared effector, policy, and demand cannot be admitted honestly.

7.2 The built-in table

The toolchain ships a default manifest of three intrinsics for the training domain. These definitions are normative for programs that rely on the defaults:

Intrinsic	Lowers to	Policy	Demand rule
`train_step()`	`train_step`	must-admit	none
`admit_checkpoint(mb=)`	`checkpoint_write`	retry	resource `storage_bw`: $mb \times mb_bw$, where `mb_bw` defaults to 9
`prefetch(depth=)`	`prefetch_read`	drop	resource `storage_bw`: $depth \times depth_bw$, where `depth_bw` defaults to 10

The policies use the kernel's three refusal meanings (Chapter 8): **must-admit** — if this operation is refused, the run aborts; **retry** — a refused operation is carried and re-proposed in later steps; **drop** — a refused operation is discarded, and the program proceeds without it.

7.3 Writing a manifest

A manifest is a TOML file of intrinsic declarations:

```
# excerpt: one intrinsic, fully declared
[[intrinsic]]
name = "admit_checkpoint"
effector = "checkpoint_write"
on_refuse = "retry"
args = ["mb"]
required_args = ["mb"]

[[intrinsic.demand]]
resource = "storage_bw"
arg = "mb"
scale = 9
```

The semantics of each field:

- `name`, `effector`: the program-side name and the runtime name it lowers to. They may be the same; they are kept distinct so a program vocabulary can stay stable while a binding's names change.
- `on_refuse`: one of `must_admit`, `retry`, `drop` (written with underscores in the manifest file).
- `args`, `required_args`: the call's named integer arguments, and which of them every call must supply. Arguments exist to compute demands; they are not passed to effectors (Chapter 6).
- `[[intrinsic.demand]]`: one computed demand — a resource name, the argument it scales, and the integer scale. Several demand entries compose; a demand the compiler cannot compute because a required argument is missing, or whose resource has no capacity recorded, is a compile error. Demands are *saturating*: an argument beyond a declared bound yields a demand at least the capacity, which the admission rule refuses rather than overflows.

Two ways to supply a manifest at compile time: **replace** (the given manifest is the whole declaration set) or **extend** (`extend_default = true` in the manifest merges its declarations over the built-in table — a name redeclared in the extending manifest replaces the built-in declaration for that name). Chapter 14's worked example uses extend mode, as real programs generally do.

© 2026 Layer1Labs Silicon Inc.

7.4 Profiles: who owns an intrinsic name

As intrinsics multiplied beyond the original training set, a flat global namespace stopped being safe: two domains can each want the same name with different demands, and a silent same-name replacement makes a compiled program's meaning depend on which manifest happened to be in force. The **profiles model** — adopted for the toolchain and being landed in the format and the compiler on the path Chapter 12 describes — replaces the flat namespace with layered ownership:

- A **core profile**, owned by the specification itself, defines the universal intrinsics every engine must provide: the operations no meaningful program runs without. (Its published identity is `chrono-core`, version 1.)
- **Domain profiles** are named, versioned sets published by the toolchain for a problem domain. The built-in table of Section 7.2 is the first of them — the GPU-training profile; the traffic program of Chapter 14 runs on the second, the traffic profile, whose four intrinsics are documented there. A domain profile's declarations override nothing outside themselves.
- **Application extensions** are namespaced intrinsics a program's own manifest declares under an application prefix. Namespaced entries are *explicitly non-portable*: a program using one runs only where that extension is provided, and the specification says so.

Under the model, a compiled specification records **which profiles it requires, at which versions**, together with a *signature digest* of each intrinsic it uses — the intrinsic's name, argument names and types, and demand resources — so that "the same name" is never again "maybe the same meaning." An engine publishes a **capability declaration**: the profiles and versions it provides. Loading compares the two (Chapter 12): a specification whose required profile, version, or signature does not match the engine's declaration is **refused at load**, with the mismatch named. The identity of a profile is written `profile/name`; a program's calls remain bare names in source text, resolved against the profiles the program declares it uses.

Status of this section. The profiles model is adopted, and its rules are stated here as the normative contract they define. Programs compiled with toolchain versions before the model lands carry the earlier form of the artifact: bare intrinsic names, no signature digests, and no recorded profile requirement — so for those artifacts the binding profile is known at compile time and is *not* recorded in the specification itself. Chapter 12 states which conformance rules apply to which artifact form, and Chapter 13 records the versions in force.

8 Execution: the kernel contract

The kernel is the part of an engine that executes a specification. Its contract is defined here independently of any engine, because both engines implement it and conformance (Chapter 12) is checked against this chapter. Where the two engines' implementations differ, Chapter 10 and Chapter 11 say so; the contract does not.

8.1 Load time

A run begins with the load sequence of Chapter 2: read the specification, validate it completely, bind an effector to every effector name, and check capabilities (under the profiles model, a specification whose requirements exceed the engine's declared capabilities is refused here). Only when all of that succeeds does step 1 become eligible to run. Nothing in load time may be deferred to run time: an engine that discovers a missing effector at step 40 has violated this contract, not encountered bad luck.

© 2026 Layer1Labs Silicon Inc.

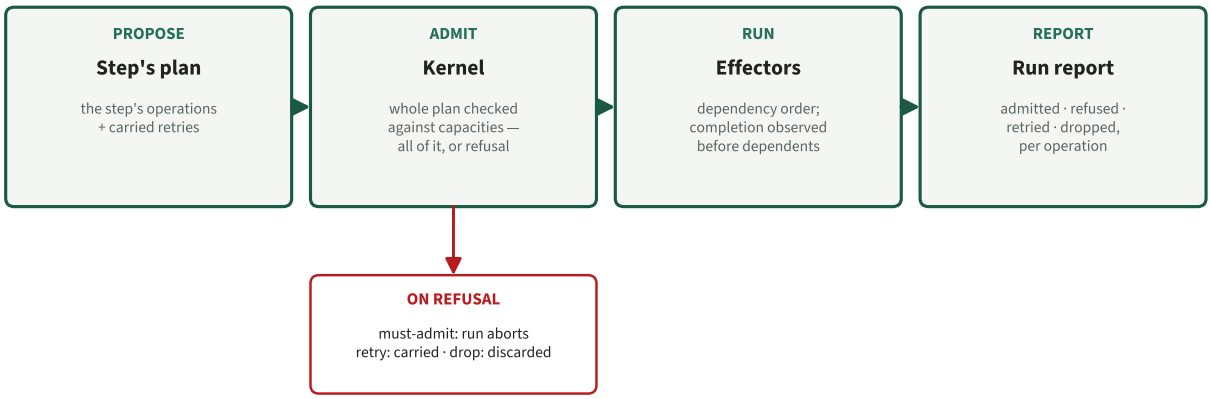


Figure 4: One plan round: propose, admit atomically, run in dependency order, report. The refusal branch is part of the round, not an exception to it.

8.2 The plan round

A **plan** is the set of operations proposed for one schedule step, plus any carried retries (Section 8.4). Each plan goes through the same round:

1. **Candidate order.** Candidates are considered in a fixed order: must-admit operations first (in schedule order), then carried retries, then fresh retry-policy operations, then drop-policy operations. The order matters — it decides which operations see capacity first — so it is part of the contract, not an implementation choice.

2. **Atomic admission.** The kernel sums the candidates' demands per resource, in candidate order, admitting each operation only if its full demand fits the remaining capacity. The whole plan's admissions are decided before any operation runs. If any must-admit operation cannot be admitted, the plan fails as a whole and the run aborts: the kernel never runs a partial plan, because the program's correctness assumed the must-admit work would happen.

3. **Execution in dependency order.** Admitted operations run in the schedule's topological order. An operation starts only after the engine has *observed completion* of every operation it depends on — a completion signal, not a timer and not an assumption.

4. **Resource lifecycle.** An admitted operation holds its demanded capacity from admission until its completion is observed; then the hold is released and the capacity returns to the pool for later steps. Holds never leak into a later step silently: an operation whose completion was never observed holds its capacity (Section 8.6), and the report shows it.

8.3 The effector contract

An effector is called with a **context** carrying what it needs to be identical across engines: the operation's identity (`op_id`, `template_id`, `effector` name), the step it runs in (`step`) and the step its operation was first proposed in (`origin_step` — different for a carried retry), the synthetic time (`now_ns`), and the attempt number (`attempt`). An effector signals success or failure; it does not return values into the kernel. The kernel's single-writer rule governs the engine side: within one engine instance, effector calls for

one plan are issued by the kernel alone, in the contract order — an engine may run independent operations concurrently only where the dependency order already permits it, and completions are recorded in observed order.

8.4 Refusals and retries

A refusal is the kernel declining to admit an operation whose demand does not fit. What happens next is the operation's policy:

- **must-admit**: the plan cannot proceed. The run aborts — honestly and immediately, with the refusing operation named in the report (Section 8.5). A must-admit refusal is a capacity fact about the deployment, surfaced at run time because that is where the capacities in force are known.
- **retry**: the operation is *carried* — it joins the next step's candidate set ahead of that step's fresh operations (after must-admits), keeping its original `origin_step`. A carried operation may be carried again, step after step, until it is admitted or the schedule ends.
- **drop**: the operation is discarded and counted. The program proceeds; that is what the policy means, and the drop is recorded, never silent.

A separate case: an operation *admitted* whose effector fails. The run continues (the admission decision was sound; the work failed), the failure is recorded against the operation, and its capacity hold is retained until an operator reconciles it — see Section 8.6.

8.5 The run report

A completed (or aborted) run returns a report with these fields:

Field	Meaning
`aborted` / `abort_reason`	Whether the run aborted, and the named reason (a must-admit refusal names the operation).
`completed_steps` / `total_steps`	Steps run to completion, against the schedule's declared length.
`admitted_ops`, `refused_ops`, `retried_ops`, `dropped_ops`	Per-operation records: identity, binding (instance path), steps, attempts, outcome.
`capacity`	The capacities actually in force for the run (the specification's, as modified at load — Section 6).
`cspec_sha256` / `lf_sha256`	The hash of the specification that ran, and the source hash recorded inside it: the report is bound to its artifact.
`chronoc_version`	The compiler version from the specification's meta section.

8.6 Edge cases, stated once

- **Drain.** If the last schedule step ends with carried retries outstanding, the kernel may run additional drain steps (proposing only the carried work) up to a configured limit — 5 extra steps by default; 0 disables draining. Drain steps are extra steps beyond the declared schedule, and the report counts them as such.

- **Failed effector, retained hold.** When an admitted operation's effector reports failure, the kernel retains the operation's capacity hold for the remainder of the run and flags the run for operator reconciliation. This is deliberate: silently freeing capacity whose real-world state is unknown (did the write happen?) would let later steps plan against a lie.
- **Time.** The only time an effector ever sees is `now_ns` = `step` × `step_ns`. Engagements that need real-world pacing (a rate limiter, a wall-clock deadline) implement it inside an effector; the kernel neither knows nor cares.
- **Determinism, restated as a testable property.** Given the same specification and the same capacities, two runs admit the same operations in the same order and record the same refusals. Effector behavior can vary (work can fail); admission cannot.

9 Extensibility: imports, packages, and libraries

Lingua Franca is a language with a module system; the Chrono target inherits it with stated limits. This chapter separates three things that are easy to conflate: what the language provides, what the subset sanctions, and what exists today.

What the language provides. In Lingua Franca v0.13.0, programs import other programs. Imports are written in the file preamble (`import X from "path/X.lf"` for a file, or an unqualified name for a package member), may be recursive, and are resolved by the compiler during its pre-generation pass: the imported reactors are read, their targets checked against the importing file's, and their parameter defaults overridden by the import statement where given. Packages — named collections of LF files distributed as units — are resolved through the same mechanism, by the Lingua Franca package tooling.

What the subset sanctions. The Chrono subset accepts imports through the same preamble channel and evaluates the result with the same rule as everything else in Chapter 4: after imports resolve, the flattened program must be fully evaluable ahead of time. An import that introduces a construct outside the subset is rejected at compile time with the construct named, exactly as if it had been written in the main file. There is no separate, permissive mode for library code.

What exists today, stated plainly. The standalone chronoc tool parses one source file; multi-file programs reach the full pipeline through the pinned `lfc` build, whose pre-generation pass resolves imports before chronoc's own analysis runs. There is, as of this manual's version, no published Chrono library: no curated set of reactors (a standard timer driver, a checkpoint coordinator) is distributed for programs to import. Programs share structure today by sharing files, not packages.

The provenance caveat. The source hash recorded in a specification (Chapter 6) covers the main source file. A program assembled from imports therefore carries a hash that does not, by itself, cover its imported files. Readers who verify provenance against the hash alone should know this; multi-file programs should record their full input set alongside the artifact until the format's provenance covers it.

Direction. The natural library story for this system — a published package of standard reactors whose operations use only core- and domain-profile intrinsics — is compatible with everything in this chapter and is future work, not a present capability. This manual will describe it when it exists.

10 Integration: the software engine (ChronoHive)

ChronoHive is the software engine: a host program loads a specification, provides Python callables as effectors, and calls the kernel to run the schedule. This chapter is the integration contract for that host program.

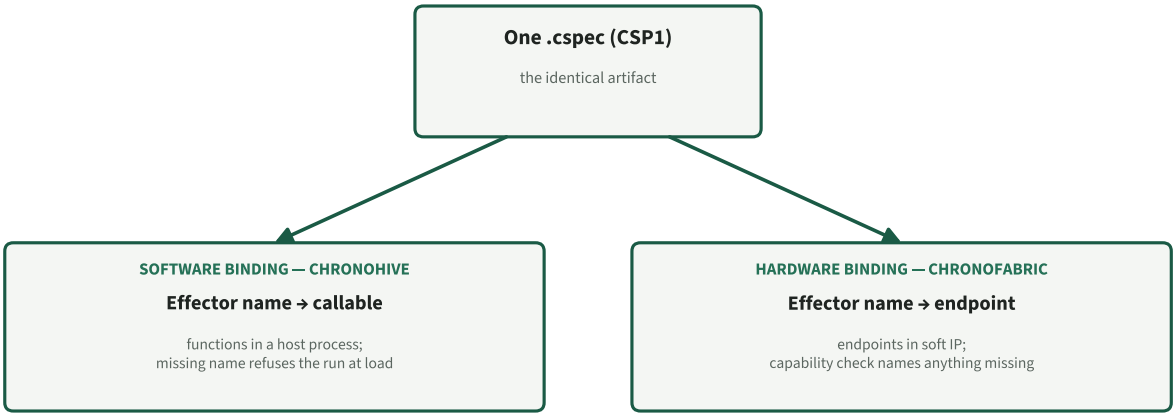


Figure 5: Two bindings, one artifact: the same `.cspec` runs under the software engine (this chapter) and under the hardware binding (Chapter 11). Only the right-hand side of each arrow differs.

The binding contract. A host provides a *binding table*: a mapping from each effector name in the specification to a callable. At load, the engine checks the table against the specification: every effector name must have exactly one callable. A name with no callable refuses the run at load — the refusal is raised before any step executes, and names the missing effector. Extra entries in the table are ignored. The callable's signature is fixed: it receives the operation context (Chapter 8) and returns nothing — success is returning; failure is raising (or returning a failure signal the host runtime defines). An effector may do anything a function may do — that is the point of the binding — but it receives no channel back into the kernel, and its return values carry no control meaning.

What the host owns. The host owns everything outside the kernel contract: constructing effectors with their real dependencies (a storage client, a model handle, a metrics sink), choosing capacities at load (Section 6 records that the report shows the capacities in force), and acting on the report — including reconciling a run flagged for a retained hold (Section 8.6).

A minimal host, in outline. In outline, a host program does five things: read the specification bytes; validate and parse them (a parse failure is an exception, and no engine exists yet); build the binding table; construct the kernel over the specification, the table, and any capacity overrides; call `run()` and keep the report. Everything a real deployment adds — concurrency around the kernel, persistence of reports, operator tooling — sits outside this outline, in the host.

Determinism at this tier. The kernel is single-writer and in-process. A host that runs several kernels concurrently runs several independent runs; determinism (Section 8.6) is a property of each run against its own specification and capacities, and the report of each run carries the hash that identifies which specification it describes.

11 Integration: the hardware tier (ChronoFabric)

ChronoFabric runs the same specification on FPGA platforms, where effectors are hardware endpoints in soft IP. The integration contract differs from Chapter 10 in exactly the places where hardware forces it to, and this chapter names each difference.

The endpoint contract. At the hardware tier an effector name binds to an endpoint of a soft-IP block. The host-side loader does not talk to endpoints directly: it presents the specification to the platform, and the platform's binding manifest states which effector names it can serve. The capability check — every

effector name in the specification must be served by the fabric's binding — runs before any load completes; an unserved name is reported, by name, as a capability error, and nothing is loaded.

The load image. The board-side artifact is a *load image*: the specification, validated on the host, lowered into the packed form the fabric consumes. The lowering is mechanical and semantics-preserving: every field of the specification — operations, demands, policies, dependencies, schedule, bindings, capacities, and the meta section including the source hash — appears in the image, packed in the fabric's layout (image format version 2 under magic `FLI1`), and the image's capacity words restate the specification's capacities field for field. A load image can therefore be checked against its specification by any reader, and a mismatch is a defect, not a tuning difference. In this version, loading an image happens in the fabric's reset state only: there is no live reconfiguration, and a new specification means a new load, in reset.

Execution at this tier. The kernel contract of Chapter 8 is the contract at this tier as well: plan rounds, atomic admission, dependency-ordered execution with observed completion, per-operation policies, and a run report. Where a measured property of the hardware differs from the software tier's — completion observed through hardware status rather than a return value — the contract's words still govern: an operation's dependents start when its completion is observed.

Published operating figures. Measured results for the fabric tier are published as their own documents in the documentation library (Appendix D), including an end-to-end demonstration in which a compiled traffic program's plans are served by a fabric endpoint under a managed service: sustained serving over a two-day soak with every request answered by a valid plan or an explicit error, and a fail-closed drill in which a deliberately broken program version received an explicit error response (HTTP 503, no plan) while the production service continued unaffected. Chapter 14 works the same program as an example. The figures in those documents are the measured record; this manual does not restate them, because a reference manual should not carry numbers that age.

Direction. The fabric roadmap — wider resource vocabularies, parameterized load images, live reconfiguration — is development direction, published as such in the fabric's own documents. Nothing in this chapter depends on it.

12 Conformance and portability

Chapter 3 promised portability in a strong sense, and a promise is only as good as its checking rules. This chapter states the rules. Each rule is marked: **[in force]** means the current toolchain and engines are checked against it today; **[adopted]** means the rule is settled and being landed with the profiles model, and artifacts or engines predating the landing are described accordingly.

For a program to be portable [in force]:

1. It compiles under the subset of Chapter 4 — anything outside the subset is not a Chrono program, on any engine.
2. It calls only intrinsics declared in manifests it ships with (or in published profiles it declares it uses); an undeclared call is a compile error everywhere, not a portability gray area.
3. Its schedule, demands, and decisions are fully determined by the specification: a program whose behavior depends on anything outside its `.cspec` file is relying on an accident of one host.

For an engine to claim conformance [in force]:

1. It reads CSP1 exactly as Chapter 6 states — total validation, reject-never-partial — including the checksum, bounds, capacity, and acyclicity checks.

2. It implements the kernel contract of Chapter 8 without substitution: candidate order, atomic admission, completion- observed dependencies, the three refusal policies with their stated outcomes, drain behavior, the retained-hold rule, and the synthetic clock.
3. It refuses, at load, any specification naming an effector it cannot bind — and names the missing effector.
4. Its run report carries the fields of Section 8.5, so a run on it can be compared with a run anywhere else.
5. Its identity claims are scoped honestly: results obtained under one engine's measurements are that engine's results; the specification's portability does not transfer performance properties, only behavior.

Under the profiles model [adopted]:

1. A specification records its required profiles and versions, and a signature digest per intrinsic used (name, argument names and types, demand resources).
2. An engine publishes a capability declaration (profiles and versions provided) and refuses at load — naming the mismatch — any specification requiring more.
3. An intrinsic's semantics are the written semantics of its profile entry, and engines are checked against them by the profiles' conformance test batteries: the same operation sequences must produce zero decision mismatches against the reference behavior. A name whose implementation differs from its profile semantics is not the same intrinsic, whatever it is called.
4. Namespaced application extensions are marked non-portable in the specification that uses them; no engine is non-conformant for declining one, and every engine must decline one it does not provide.

The artifact-form note from Section 7.4 applies here in full: specifications in the current (pre-profile) form carry bare names and no digests, and for them the [in force] rules are the whole conformance story. The [adopted] rules govern artifacts and engines as the model lands; a document claiming conformance for a pre-profile artifact under the profile rules would be claiming evidence the artifact does not contain.

13 Pins, versions, and provenance

Everything in this manual is pinned. This chapter is the pin list; if a statement anywhere else in the manual and a pin here ever disagree, the pin wins and the other statement is a defect.

© 2026 Layer1Labs Silicon Inc.

Item	Pin
Language baseline	Lingua Franca **v0.13.0**. Upstream publishes no separately versioned language specification; the baseline is the v0.13.0 release tag and the Lingua Franca handbook as of that release.
Compiler build providing the target	Reports itself as `lfc 0.14.0-SNAPSHOT` (a post-0.13.0 development build of the Lingua Franca repository). The stock-compiler validation gate is run at 0.13.0.
chronoc	`0.1.0`. Toolchain releases are the governance record for the compiler's version; the version string recorded inside a given `.cspec` artifact's meta section is whatever the producing build wrote there — read the artifact, not this line, for a specific file.
Specification format	CSP1, format version 1 (Chapter 6).
Fabric load image	Format version 2, magic `FLI1` (Chapter 11).
This manual	Version 1.0, October 27, 2026. Living companion: Chapter 12's [adopted] rules land with the profiles model, and the manual is revised as they do.

On version strings generally: this toolchain's tools report their versions in the forms their own ecosystems use, and the forms are not uniform (a development-snapshot string for the compiler build; a semantic version for chronoc). Provenance lives in the artifacts: a specification's meta section records the source hash and the versions that produced it, and a run report binds itself to the specification by hash (Section 8.5). A claim about a specific run should cite those recorded values, not this chapter.

14 Worked example: TrafficGrid

This chapter reads one real program end to end. TrafficGrid is a traffic-signal timing program: nine intersections on a corridor, timed together. It is Layer1Labs' published cross-tier example — the same program whose compiled specification runs under the software engine and, on the hardware tier, behind a managed service whose measured record is published separately (Appendix D). Every number in this chapter is from the published program and its compiled specification.

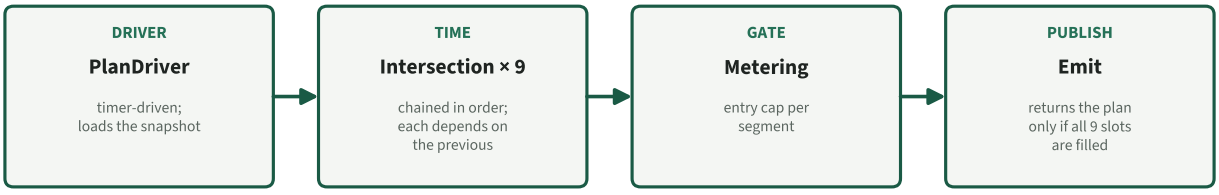


Figure 6: TrafficGrid's reactor chain: one program round is one schedule step, and the step's 12 operations are admitted atomically.

14.1 The program

The program is a chain of reactors driven by one timer:

- **PlanDriver** holds the driver reaction (Chapter 4): a timer stepping `cycles` rounds, an intrinsic that loads the traffic snapshot, and the stop guard at the declared horizon.
- **Intersection**, instantiated nine times and chained: each intersection receives the previous intersection's plan state on a port, advances the corridor's timing by one intersection, and passes its state onward. The chaining is ordinary LF wiring; in the compiled specification it becomes the dependency order (Chapter 5).
- **Metering** gates entries per road segment against a declared per-segment cap.
- **Emit** publishes the round's plan — but only if every one of the nine intersections contributed. A partial plan is never emitted; the reaction counts the filled slots and publishes nothing when the count falls short.

Call-site constants carry the program's tuning in the open: cycle bounds (36–72), a clearance interval (2), clamps for queue and ETA values (999 and 9999), thresholds for left-turn pressure (3, with a crossing term 24), a travel-time constant (116), and the entry-segment cap (20). These are program parameters in the sense of Chapter 5: resolved at compile time, visible in the source, identical on every engine.

14.2 The intrinsics

The program's four intrinsics are declared in its own manifest, which *extends* the built-in table (Chapter 7) rather than replacing it:

Intrinsic	Policy	Meaning
`sense_snapshot`	must-admit	Load the snapshot the round plans from.
`time_intersection`	must-admit	Advance one intersection's timing; demanded against the `planner` resource at (units $\times$ 1) per call, with a built `planner` capacity of 9.
`meter_entries`	must-admit	Gate segment entries against the cap.
`emit_plan`	must-admit	Publish the completed plan.

All four are must-admit: a round that cannot be fully planned is not a degraded round, it is no round — and the kernel contract (Chapter 8) turns that policy into behavior: if any of the round's operations cannot be admitted, the plan fails atomically and no partial plan exists to publish. Under the profiles model of Section 7.4, these four are the traffic profile's intrinsics.

14.3 The compiled specification

Compiled at its published configuration, the program's specification contains **12 operations** in a **1-step** schedule (each service request runs one round) and is **1,563 bytes** — small enough to print, which is the point of showing it: nine `time_intersection` operations (one per intersection, dependency-chained), the snapshot load, the metering operation, and the emit. The planner capacity of 9 against nine unit demands is the arithmetic of the whole design in miniature: the step's combined demand exactly fills the capacity, so admission succeeds by construction, and any program variant that added a tenth intersection against the same capacity would be refused rather than over-subscribed.

© 2026 Layer1Labs Silicon Inc.

14.4 Reading the example back through the manual

- The subset (Chapter 4): one main reactor, timer-driven, counter-bounded; the chained reactors are instances; the intrinsics are the only calls.
- Unrolling (Chapter 5): the horizon is one round per request, so the schedule is one step; the chain wiring is the dependency order.
- The artifact (Chapter 6): 1,563 bytes, all eight sections, nothing else. No timing arithmetic is in the file — the arithmetic is the effectors' implementation, bound by name.
- Execution (Chapter 8): one plan round per request, atomic admission against the planner capacity, completions observed down the chain, a report per run.
- The boundary (Chapter 3): nothing in the program, the manifest, or the specification says *how* an intersection is timed. That is the effector provider's implementation — on the software tier a function, on the hardware tier an endpoint — and it is checked by Chapter 12's rules, not by reading this program.

14.5 The example in service

The published demonstration of this program runs the hardware binding behind a managed HTTP service: a request carries a traffic snapshot; the service runs one round of the compiled specification on the fabric endpoint and returns the plan. The measured record — sustained serving over a 48-hour soak, and a fail-closed drill in which a deliberately broken version of the program was refused with an explicit error while production traffic continued — is published in the ChronoFabric measured-results document (Appendix D) and is not restated here (Chapter 11). For this manual, the example's role is the one this chapter used it for: a program small enough to hold in the hand, exercising every rule in the book.

A Manifest quick reference

A manifest is a TOML file. One intrinsic per `[[intrinsic]]` table; one demand per `[[intrinsic.demand]]` table under it.

Field	Where	Meaning
`extend_default`	file top	`true`: merge over the built-in table (a re-declared name replaces the built-in). Absent/false: the file is the whole declaration set.
`name`	intrinsic	The program-side name programs call.
`effector`	intrinsic	The runtime name engines bind.
`on_refuse`	intrinsic	`must_admit` / `retry` / `drop`.
`args`	intrinsic	Named integer arguments the call may take.
`required_args`	intrinsic	Arguments every call must supply.
`resource`	demand	The budgeted resource the demand counts against.
`arg`	demand	The argument the demand scales.
`scale`	demand	Integer multiplier: demand = arg × scale. Saturating (Section 7.3).

Worked declarations: the built-in table (Section 7.2) and the traffic manifest (Section 14.2). Compile-time behavior for a malformed manifest is in Appendix B.

B Error and refusal behavior

Every failure in this system is either a **compile error** (the program never becomes a specification), a **load refusal** (the specification never runs), or a **run-time refusal** (an operation is declined under its policy). Nothing fails silently; each entry below names where it is recorded.

Compile errors (no specification is produced): a program the validation gate rejects (grammar, or a construct outside the Chapter 4 subset — the error names the construct); a missing or malformed driver reaction or stop bound (including a bound outside 1..1,000,000); a call to an undeclared intrinsic; a manifest argument missing where required, or a demand whose resource has no capacity; a dependency cycle in the wiring; a quantity that cannot be reduced to a compile-time integer; `workers` > 0.

Load refusals (no step runs): a specification failing any Chapter 6 validation (bad magic or version, checksum mismatch, out-of-bounds index, a single demand exceeding its capacity, cyclic dependencies) — the reader names the failed check; an effector name with no binding (software) or no serving endpoint (hardware) — named; under the profiles model, a required profile, version, or signature digest exceeding the engine's published capabilities — named (rule marked [adopted], Chapter 12).

Run-time behavior: a refused must-admit operation aborts the run and is named in the report (`abort_reason`); retry-policy operations are carried per Section 8.4 and appear in `retried_ops`; drop-policy operations are counted in `dropped_ops`; an admitted operation whose effector fails is recorded against the operation, its capacity hold is retained, and the run is flagged for operator reconciliation (Section 8.6). A service hosting the engine maps a run-level refusal to its clients as an explicit error (the published demonstration used HTTP 503) — never a partial result presented as success.

C Glossary

Admission The kernel's decision, made for a whole plan before any of it runs, that an operation's demands fit the capacities in force.

Binding (1) The record tying an operation to the reactor instance that proposed it. (2) The table (software) or endpoint map (hardware) tying effector names to implementations.

Capacity The budget of one resource, in integer units, recorded in the specification and possibly lowered at load.

chronoc The compiler from a Chrono-target Lingua Franca program to a constraint specification.

ChronoHive The software engine: effectors are functions in a host process.

ChronoFabric The hardware tier: effectors are endpoints in soft IP on FPGA platforms.

Constraint specification (.cspec) The compiled artifact: operations, demands, dependencies, capacities, and schedule, in format CSP1.

Demand The integer amount of one resource an operation requires, computed at compile time from the manifest's rule.

Driver reaction The one timer-triggered reaction that increments the program's counter and stops it at the declared bound.

© 2026 Layer1Labs Silicon Inc.

Effector The implementation behind an effector name: a callable (software) or an endpoint (hardware).

Intrinsic A named operation a program may call, completed by an effector manifest declaration.

Kernel The part of an engine that executes a specification under the Chapter 8 contract.

Load image The hardware tier's packed form of a specification (FLI1, version 2), semantics preserved field for field.

Plan The operations proposed for one step, plus carried retries — the unit of atomic admission.

Policy / refusal policy What a refusal means for an operation: must-admit, retry, or drop.

Profile A named, versioned set of intrinsics: the core profile, domain profiles, or a namespaced application extension.

Run report The record a run returns: admissions, refusals, retries, drops, capacities in force, and the specification's hash.

Step One position in the unrolled schedule; one plan round.

Subset The Chapter 4 rules defining which Lingua Franca programs are Chrono programs.

Template One distinct intrinsic call site in a program, identified by a template ID in the specification.

D Companion documents

This manual is one document in the Layer1Labs library. The companions below are published alongside it at `https://layer1labs.ai/docs/`:

- **ChronoHive Architecture Overview** — the software platform's architecture in brief, for readers who want the shape before the reference.
- **ChronoHive Deep Dive: The Admission-Control Engine** — the engine's mechanism and measured behavior, in depth.
- **ChronoHive Measure Results** — the software engine's measured record.
- **ChronoFabric Architecture Overview** and **ChronoFabric Technical Brief** — the hardware tier's architecture and implementation position.
- **ChronoFabric Measure Results** — the hardware tier's measured record, including the TrafficGrid service results this manual's Chapter 14 points to.
- **Evaluating ChronoFabric: Beta Participation and Licensing** — the hardware tier's beta participation and licensing process.
- **Applying ChronoHive Admission Control to Existing Storage Workflows** and **Evaluating ChronoHive: The Pilot Process** — implementation and evaluation process companions.

Where a companion and this manual treat the same subject, this manual is the reference for language, format, and contract statements, and the companions are the record for measurements and program process.

© 2026 Layer1Labs Silicon Inc.

Version history

Version	Date	Notes
1.0	2026-10-27	First issue.

© 2026 Layer1Labs Silicon Inc.

Notices

Copyright. © 2026 Layer1Labs Silicon Inc. All rights reserved.

Sharing. This document may be shared in its complete, unmodified form, with attribution to Layer1Labs Silicon Inc. No other reproduction, modification, adaptation, or derivative use — in whole or in part — is permitted without the prior written permission of Layer1Labs Silicon Inc.

No license. Nothing in this document grants, by implication or otherwise, any license or right under any patent, copyright, trademark, or other intellectual property of Layer1Labs Silicon Inc. Publication of this document is a disclosure of information only.

Patents. The technology described in this document is the subject of pending patent applications, including 9 provisional patent applications filed with the U.S. Patent and Trademark Office. No statement in this document is a representation as to the scope, validity, or enforceability of any application or of any patent that may issue from one.

Trademarks. Layer1Labs, ChronoHive, and ChronoFabric are trademarks of Layer1Labs Silicon Inc. Other names mentioned in this document may be trademarks of their respective owners; their use here is for identification only and claims no affiliation or endorsement. In particular, Lingua Franca is an open-source research project of its own community.

Measurements and basis. Figures in this document are drawn from the project's measurement records and are cited where they appear; where a figure is modeled rather than measured, it is labeled as modeled. This document is provided for information only, on an "as is" basis, without warranty of any kind, express or implied, including as to accuracy, completeness, or fitness for a particular purpose. It describes a demonstration system; it is not a specification, an offer, or a forecast of results in any other setting.

Current version. Documents in this family are revised in place under version numbers. The current version of this document is the one published at layer1labs.ai; earlier versions are superseded.