Makeline System

Hot-reloadable configuration for food assembly hardware

Quick Start

```
# Initial generation
just generate-makeline simulation
just simulate simulation

# For hot-reload: edit generated makeline.json
# Edit profiles/simulation/makeline.json
# Ctrl+S → reloads in 2.5s (no regeneration needed)
```

Aliases: `gm` / `s`

Always use `just generate-makeline` (not `just generate`)

Generation Modes

`just generate` (machine_config mode):

- Spawner entries WITHOUT `-M` flag
- Modules read config from `config.json` file
- Communication via ConfigTopic (file-based)
- Uses Identity adapter

`just generate-makeline` (makeline_server mode):

- Spawner entries WITH `-M` flag
- Modules query `makeline_server` for config

Where Files Live

```
generated/profiles/simulation/
├─ simulation.json          # Don't edit (gets regenerated)
├─ spawner.json             # Process launch config
├─ config.json              # Machine config
└─ watch.json               # File watch config

profiles/                   # Source + generated
├─ simulation.json          # Source profile (edit for full regen)
└─ simulation/
   ├─ makeline.json         # Generated (edit for hot-reload)
   └─ backups/              # Timestamped copies
```

For hot-reload: Edit `profiles/simulation/makeline.json` | **For full regen**: Edit `profiles/simulation.json` then run `just generate-makeline`

Profile Structure

Three sections define your makeline:

layouts: CabinetKind list (hardware topology)

- `{ "default": { "cabinets": ["Initial", "Denest", "Dispense", "Lift"] }}`

layer_groups: Named edit collections (modifications)

- `{ "base": [layer1, layer2], "prod": [layer3] }`
- Layers applied sequentially (order matters!)

line_builds: Combine layout + layer groups (final config)

Two Workflows - Part 1

Workflow A: Full Regeneration

- Edit `profiles/simulation.json` (source profile)
- Run `just generate-makeline simulation`
- Outputs to `profiles/simulation/makeline.json`
- Use when: Changing layouts, layer_groups, line_builds structure

Workflow B: Hot-Reload

- Edit `profiles/simulation/makeline.json` (generated file)
- Save → 2.5s → Changed modules reload

Two Workflows - Part 2

What happens during hot-reload:

- Diff new vs old makeline.json
- Backup to `backups/{timestamp}/`
- Send SectionChanged events
- Reload affected modules only

What keeps running: Planner, unchanged modules, active orders

Common Tasks - Part 1

Change dispenser ingredient:

```
{ "EditSectionField": {
  "identity": { "owner": "dispenser-3", "subject": "self" },
  "section_name": "inputs",
  "field_key": "assigned_ingredient_id",
  "field_value": "black_beans"
}}
```

Adjust buffer motion timeout:

```
{ "EditSectionField": {
  "identity": { "owner": "buffer-1", "subject": "self" },
  "section_name": "configuration",
```

Common Tasks - Part 2

Test without HVAC module (for debugging):

```
{ "OmitModules": {
  "identities": [{ "owner": "hvac-1", "subject": "self" }]
}}
```

To apply these edits:

- Add to layer in source profile → run `just generate-makeline` (full regen)
- Or directly edit module sections in `profiles/simulation/makeline.json` → hot-reload in 2.5s

Physical Hardware Context - Part 1

Real-world machine structure drives software design:

Cabinet = Physical enclosure unit

- Initial: System computer, no food hardware
- Denest: Unstacks bowls from dispenser
- Dispense: Contains ingredient hoppers (12-18 per cabinet)
- Lift: Presents finished bowls to customer

Device = Functional hardware subsystem within cabinet

- Core: Software-only (system services)

Physical Hardware Context - Part 2

Module = Control software for specific hardware

- Buffer: Manages bowl staging area (motor + position sensors)
- Dispenser: Controls auger motor, reads RFID tag, weighs portions
- Lift: Raises/lowers bowl presentation platform
- Planner: Plans bowl assembly sequence
- Follower: Tracks bowl position throughout system

Physical flow: Bowl enters Denest → unstacked → moved to Dispense cabinet → positioned under dispensers → ingredients added → moved to Lift → presented to customer

System Tools - Part 1

makeline_generator (binary):

- Reads source profile, expands layout, applies layers, outputs files
- Run via: `just generate-makeline simulation`

spawner (binary):

- Process manager that launches all modules
- Reads `generated/profiles/simulation/spawner.json`
- Run via: `just simulate simulation`

System Tools - Part 2

makeline_server (binary):

- Config provider for modules in `-M` mode
- Watches `profiles/simulation/makeline.json` for changes
- Performs diff and sends SectionChanged events

makeline (module):

- One of 21 Core modules (NOT makeline_server)
- Coordinates system behavior

Explorer (tool):

Architecture Rationale - Part 1

Hardware abstraction: Physical machines have cabinets → devices → modules. Software mirrors this hierarchy so config matches reality.

Hot-reload requirement: Must change config without stopping production

- Graph-based diff detection (compare old vs new)
- Selective module restart (only changed modules)
- Process isolation (spawner manages independent processes)

Architecture Rationale - Part 2

Distributed system: Each module = separate process via IPC

- Fault isolation: One crash doesn't kill entire system
- Independent scaling: Can run modules on different machines

Configuration flexibility: Layers allow base config, env overrides, dev tweaks, runtime line build switching

Graph structure: Represents parent-child relationships, traversable for planning/validation/visualization

System Architecture

Three-tier enum hierarchy: CabinetKind → DeviceKind → ModuleKind

Generator expands each tier: Types define requirements → instances get created

CabinetKind enum (4 variants):

- `Initial`, `Denest`, `Dispense`, `Lift`

DeviceKind enum (10 variants):

- `Core`, `CabinetCore`, `CabinetScreen`, `Denester`, `Dispenser`

Cabinet → Device Mappings

Initial cabinet (system-wide services):

- `Core` device → 21 modules

Denest cabinet (bowl handling):

- `CabinetCore`, `Conveyance`, `Denester`, `CabinetScreen`

Dispense cabinet (ingredient dispensing):

- `CabinetCore`, `Hvac`, `DispenseFillPositioner`
- **Note**: Dispenser devices added separately (see Dispenser Special Case)

Dispenser Special Case - Part 1

Dispensers don't follow standard enum expansion

Standard expansion: `cabinet.devices()` returns DeviceKind list → each device expanded to modules

Dispenser expansion: NOT in `Dispense.devices()` return value

- Added dynamically via `AssignDispensers` layer edit during layer application
- Why: Variable count (12 for v5111, 18 for v5112), per-dispenser configuration

Dispenser Special Case - Part 2

Process: Layer application calls graph mutation functions

- `makeline.add_device_to_cabinet(DeviceKind::Dispenser, cabinet_node)`
- `makeline.add_module_kind_to_device(ModuleKind::Dispenser, device_node)`
- Applies DispenserAssignment config (ingredient_id, position, kind)

Result: Dispensers are layer-configured, not layout-defined (exception to enum expansion pattern)

Device → Module Details

Core device (Initial cabinet only) - 21 system modules:

- `Api-1`, `BowlRecovery-1`, `CabinetMonitor-1`, `Datalog-1`
- `Discovery-1`, `Echo-1`, `Fault-1`, `Follower-1`
- `Interlock-1`, `LifeCycler-1`, `MachineConfig-1`, `Makeline-1`
- `PartnerApi-1`, `PartnerWebhook-1`, `Planner-1`, `Preprocessor-1`
- `RfidClient-1`, `Sequencer-1`, `State-1`, `Telemetry-1`, `Tracker-1`

DispenseFillPositioner device (one per dispense cabinet):

- `Buffer`, `Conveyance`, `Shutter`, `Duc`

Core Modules Overview - Part 1

System orchestration:

- Planner: Assembly planning (which dispenser, what order)
- Sequencer: Executes plans as bowl moves through system
- Follower: Tracks individual bowls (position, state)

Hardware control:

- Makeline: Central coordinator, config hot-reload
- MachineConfig: Provides config to other modules
- State: System state machine (Idle, Running, Faulted)

Core Modules Overview - Part 2

Integration:

- Api: REST API for external systems
- PartnerApi, PartnerWebhook: Partner integrations
- Telemetry: Metrics collection and reporting
- Datalog: Event logging to database

Fault handling:

- Fault: Fault aggregation and display
- Interlock: Safety interlocks (door sensors, emergency stop)

Graph Structure Deep-Dive - Part 1

Graph representation: Directed acyclic graph (DAG)

Node types:

- Root (single)
- Cabinet(CabinetKind) (1-4 nodes)
- Device(DeviceKind) (variable count)
- Module(module_data) (40-100+ nodes)

Edges: Parent → child relationships

- Root → Cabinets

Graph Structure Deep-Dive - Part 2

Why parent-child?

- Config inheritance: Children reference parent config
- Logical grouping: Modules in device share context
- Traversal: Walk graph to find modules by type/location
- Validation: Ensure required modules exist

Graph traversal uses:

- Planner: Find all dispensers in cabinet 2
- Config validation: Ensure each cabinet has required devices

Generator Expansion Process - Part 1

Step 1: Read `profiles/{preset}.json` → get layouts, layer_groups, line_builds

Step 2: Select line_build (from CLI or "default") → determines layout + which layer_groups to apply

Step 3: Expand layout into graph:

```
For each CabinetKind:
  cabinet_kind.devices() → Vec<DeviceKind>
  For each DeviceKind:
    device_kind.modules() → Vec<ModuleKind>
    Create numbered instances (buffer-1, buffer-2, ...)
```

Generator Expansion Process - Part 2

Step 4: Apply layers sequentially (order matters!):

- Each layer contains edits (EditSectionField, AssignSections, etc.)
- Later layers override earlier layers
- Edits target specific modules by Identity

Step 5: Write output files:

- `profiles/{preset}/makeline.json` - Expanded graph (watched for hot-reload)
- `generated/{preset}/simulation.json` - Copy of source (don't edit)

Module Sections Reference

Different modules have different section names:

Buffer modules: `configuration`

- Fields: motion_timeout_ms, homing_velocity, home_to_lower_mrad

Dispenser modules: `inputs`, `outputs`

- inputs: assigned_ingredient_id, dispenser_kind
- outputs: Runtime state (read-only)

Lifecycler module: `configuration`, `light`

Layer Edit Types - Part 1

EditSectionField: Change single config field (most common)

```
{ "EditSectionField": {
  "identity": { "owner": "buffer-1", "subject": "self" },
  "section_name": "configuration",
  "field_key": "motion_timeout_ms",
  "field_value": 20000
}}
```

AssignSections: Replace entire sections (multiple related fields)

```
{ "AssignSections": {
  "identity": { "owner": "lifecycler-1", "subject": "self" },
  "sections": {
    "configuration": { "cooldown_complete_ms": 24000, "timeout_fault_ms": 600000 },
```

Layer Edit Types - Part 2

AssignDispensers: Populate all dispensers for cabinets

```
{ "AssignDispensers": {
  "dispensers": [
    { "cabinet_index": 2, "dispenser_index": 0, "ingredient_id": "black_beans",
      "position": { "x": 0.0, "y": 0.0, "z": 0.0 }, "kind": "Auger" },
    { "cabinet_index": 2, "dispenser_index": 1, "ingredient_id": "rice",
      "position": { "x": 0.1, "y": 0.0, "z": 0.0 }, "kind": "Auger" }
  ]
}}
```

Layer Edit Types - Part 3

OmitModules: Remove modules from graph (testing/debugging)

```
{ "OmitModules": { "identities": [
  { "owner": "hvac-1", "subject": "self" },
  { "owner": "dispenser-5", "subject": "self" }
]}}
```

AssignGlobalConfiguration: Set global makeline config

```
{ "AssignGlobalConfiguration": {
  "assignment": { "OperationMode": "Production" }
}}
```

Module Communication - Part 1

IPC via ZeroMQ: Each module = separate process on pub/sub network

Identity for routing: Messages addressed by `{ owner, subject }`:

- `{ owner: "buffer-1", subject: "self" }` → message to buffer-1 process
- `{ owner: "buffer-1", subject: "motor-1" }` → message to buffer-1's motor child
- makeline_server routes based on Identity

Message types:

Module Communication - Part 2

Why Identity matters:

- Layers target modules by Identity
- IPC routing uses Identity
- Planner sequences using Identity
- Logging/debugging traces by Identity

Contract-based: Each module type has typed contracts (buffer_contract, dispense_contract, etc.)

How Targeting Works - Part 1

Identity structure: `{ owner: "module-name", subject: "target" }`

owner: Module instance name (numbered instances from graph expansion)

- `"buffer-1"`, `"buffer-2"` (dispense cabinet buffers)
- `"lifecycler-1"` (system singleton)
- `"lift-1"` (lift cabinet)
- `"dispenser-1"` through `"dispenser-12"` (v5111) or `"dispenser-18"` (v5112)

subject: Config target within the module hierarchy

How Targeting Works - Part 2

Module hierarchy examples:

`buffer-1` has children: `motor-1`, `home-sensor`, `drip-tray`, `time-of-flight`

- Edit buffer config: `{ owner: "buffer-1", subject: "self" }`
- Edit buffer motor: `{ owner: "buffer-1", subject: "motor-1" }`

`dispenser-3` has children: `motor`, `rfid`

- Edit dispenser config:
  `{ owner: "dispenser-3", "subject": "self" }`

Hot-Reload Mechanics - Part 1

What triggers reload? Edit `profiles/{preset}/makeline.json` → makeline_server detects → reloads and diffs

Diff algorithm:

1. Load new profile, expand to graph
2. Compare new graph vs old graph (structure + sections)
3. Identify changed modules (section values differ)
4. Send SectionChanged events to affected modules
5. Modules reconfigure without restarting process

Hot-Reload Mechanics - Part 2

What causes module restart? (spawner action)

- Module added/removed from graph
- Module kind changed
- Executable path changed

What's hot-reloadable?

- Section field values (timeouts, ingredients, positions)
- Section addition/removal
- Child module config

Custom Layers

Dev tweaks without modifying preset profiles

`just generate-makeline-custom simulation` → creates `custom_layers.json`

```
[{
  "metadata": { "name": "Faster Buffer Motion" },
  "edits": [{
    "EditSectionField": {
      "identity": { "owner": "buffer-1", "subject": "self" },
      "section_name": "configuration",
      "field_key": "motion_timeout_ms",
      "field_value": 10000
    }
  }]
```

Line Builds & Switching

One profile, multiple configs via different layer combinations

```
"line_builds": {
  "production": { "layer_groups": ["base", "prod_ingredients"] },
  "testing": { "layer_groups": ["base", "test_ingredients"] }
}
```

At generation: `just generate-makeline simulation testing`

At runtime (Explorer):

- `AvailableLineBuilds` → see options
- `SelectLineBuild { line_build_name: "production" }` → switch (10-

Generated Files Explained

spawner.json: Process launch configuration

- Executable paths, args, environment variables for each module
- `generate-makeline`: Includes makeline_server references (enables hot-reload)
- `generate`: Includes machine_config references only (no hot-reload)

makeline.json: Expanded module graph

- Lives in `profiles/{preset}/makeline.json` (generated here, not in generated/ dir)
- Contains all module instances with their sections

- Adds hot-reload support via makeline_server in spawner.json

Watch mechanism: `profiles/{preset}/makeline.json` triggers 2.5s auto-reload

Spawner behavior: Smart restart - only kills/restarts processes with config changes

Never edit: `generated/` directory - gets overwritten on every generation

Available presets:

- `simulation` - Mock hardware, no real devices
- `y5111` - 4 cabinets, 12 dispensers

That's It

```
just generate-makeline simulation [line_build]
just simulate simulation
# Edit profiles/simulation/makeline.json for hot-reload
```

Key features:

- Hot-reload (2.5s, no restarts)
- Custom layers (dev tweaks)
- Line builds (test/prod switching)
- Auto backups (rollback ready)

Remember: Edit `profiles/{preset}/makeline.json` for hot-reload.