Integration Guide

Fiscal Printer Service — Documentation

Integration Docs

This document is the primary entry point for developers integrating with the Fiscal Printer Service over HTTP. It provides a high-level overview of the API, outlines key constraints and integration flows, explains the error model, and points to the OpenAPI contract for the complete interface definition.

It is intended for developers consuming the REST API, integrators building POS or device-management clients, and QA teams validating client-server interactions.

Start Here

First Integration Path

If you are integrating a new client, use this order:

1. Read API Overview to understand what the device exposes and which endpoints are part of the normal integration surface.
2. Read Service Configuration to see which parameters exist and which of them come from `/config` versus startup JSON.
3. Read Constraints and Behavior before writing client logic, because many operations are valid only in specific fiscal, registration, or printer states.
4. Read Integration Flows to understand the normal lifecycle: setup, registration, daily fiscal work, and reports.
5. Read Exports when implementing KLEN or fiscal-memory download workflows.
6. Use Local Simulation as the default local integration environment before moving to a real device.
7. Use OpenAPI Reference Guide and the Runtime Swagger/OpenAPI Reference for exact request and response details.
8. Keep Errors and Status Codes nearby while implementing retries, error handling, and operator-facing diagnostics.

Source of Truth Rules

- `/resources/api` is the runtime-friendly browser entrypoint for the Swagger/OpenAPI reference on a running device.
- `doc/swagger.yml` is the canonical endpoint contract artifact.
- The Markdown files in `doc/public/` explain how to use the service and what to watch out for.
- These docs should not duplicate a full endpoint-by-endpoint reference that can drift away from Swagger.

Main Areas of the API

Most integrations usually touch three areas:

- local fiscal REST API This is the primary integration surface for setup, sales, reports, journal reads, and normal device operation.

- NRA-related flows exposed through the service These are used for registration, registration reconciliation, and support/investigation around NRA communication.
- firmware/update behavior Updates are driven by the device itself and applied by the boot-time installer. The endpoints that manage them are device-management tooling, not part of the published client contract.

Public Versus Operational Surface

- Public integration surface: setup/configuration endpoints, sales endpoints, reporting endpoints, journal reads, and the normal registration flow
- Operational or internal-facing surface: diagnostic, lab and machine-to-machine endpoints (device troubleshooting, raw NRA traffic, modem control, log access). These are **not** listed in the published OpenAPI spec or in these docs; they exist on the device but are not a supported integration contract
- When in doubt, treat Swagger endpoint descriptions and the guidance in api-overview.md and constraints-and-behavior.md as the source for whether an endpoint belongs in the normal client workflow

Recommended Development Environment

- For most client development, use the Docker simulation environment first.
- The simulator packages the local fiscal REST API together with an NRA mock server and is the recommended default for the large majority of integration work.
- Real hardware should be used later for physical-printer, device-specific, and final acceptance validation.
- Detailed build and run instructions stay in the repository under `tools/README.md` and `tools/docker-sim/README.md`.

Related Docs

- OpenAPI Reference Guide
- API Overview
- Service Configuration
- Runtime Swagger/OpenAPI Reference

Owner: Backend/Embedded team Last updated: 2026-05-19

API Overview

Purpose

This section gives API consumers a compact overview of what the service exposes and where to find the authoritative API contract.

Scope

This section is limited to high-level API surface orientation. Detailed request and response definitions are provided in `doc/swagger.yml` and in the runtime Swagger/OpenAPI reference available at /resources/api.

Audience

This section is intended for:

- Developers starting a new integration
- Reviewers who need to understand the API surface quickly

What The Service Exposes

- Sales and receipt operations
- Device setup and provisioning
- Registration and NRA-related flows
- Reports, journal access, and exports
- Firmware and system-management endpoints
- Master-data CRUD for local configuration entities

What An Integrator Can Do With The Device

This API is intended for developers integrating the fiscal device into:

- point-of-sale software
- petrol-station software
- or other business applications that need fiscal device support

The goal of the integration is to ensure that when a sale or another fiscal operation is performed in the business software, the corresponding operation is also executed on the fiscal device, reported where required, and a receipt or other fiscal document is produced when needed.

In practical terms, the REST API exposes the operations required to represent and operate the fiscal device from a larger sales or retail application.

What Is Done Before Integration Starts

The device is provisioned in advance by the vendor/service organization.

Provisioning is not a normal integrator operation. It is performed in service/factory conditions and initializes:

- fiscal memory
- KLEN/local persistent state

After that initial provisioning step, the integrator can use the REST API for normal device setup and day-to-day fiscal operations.

Standard Integration Lifecycle

The standard device lifecycle for an integrator is:

1. **Provisioning**
    - performed in service/factory conditions
    - not available as a normal integrator operation
2. **Customer setup / local configuration**
    - set VAT configuration
    - configure device settings
    - configure company/header/cliche information
    - configure payment labels where needed by regulation
    - configure date and time
3. **Registration with NRA**
    - register the device through the NRA-related API flow
4. **Normal daily operations**
    - sales
    - storno operations
    - duplicates
    - other day-to-day fiscal operations supported by the API
5. **Reports**
    - X reports
    - Z reports

What An Integrator Can And Cannot Do

Can do

- configure the device after provisioning
- program VAT and other required configuration
- perform registration-related flows through the supported API
- execute normal fiscal operations
- request operational and fiscal reports
- build a client application that treats the device as a REST-connected fiscal printer/server

Cannot or should not do

- perform factory/service provisioning as part of the normal integration flow
- treat all endpoints as generic unrestricted CRUD
- ignore the fiscal/device-state restrictions attached to many operations

In other words:

- Swagger and the runtime reference at /resources/api tell the integrator how to call the API
- the integration documentation explains when those operations are valid and where they belong in the device lifecycle

Practical API Surface Matrix

The API surface is easier to understand if it is split by normal usage intent.

Setup-only or pre-operation surface

These endpoints are typically used during customer setup, device preparation, or controlled maintenance windows:

- `/header`
- `/vat`
- `/config`
- `/payment/labels`
- date-time endpoint
- `/registration`
- `/change-registration`
- `/deactivate-registration`

Integrator note:

- many of these operations are state-dependent
- they should not be treated as arbitrary day-to-day CRUD once the device is already in active fiscal use

Device KLEN mode and replacing a KLEN

`/system/info` reports a `klenMode` field describing the relationship between the attached KLEN (the local SQLite database) and the device's fiscal memory:

- `NORMAL` — the attached KLEN matches fiscal memory; the device operates normally.
- `READONLY` — a different (foreign) KLEN with existing data is attached. Fiscal write operations (`/sell`, `/storno`, `/cashInOut`, `/vat`, registration changes, `/report/Z_REPORT`, `/report/X_REPORT`) are rejected with **HTTP 423**. Reports and queries still work, but are not journaled (numbers do not advance).
- `NEEDS_INIT` — a fresh/blank KLEN is attached on a provisioned device. Fiscal writes are blocked the same way, and the device prints a "new KLEN — initialization required" notice at startup.

Bringing a replaced (blank) KLEN back into service is a **service operation**: the new KLEN is initialized from the data recoverable out of fiscal memory and the NRA (registration, VAT, last Z report number and accumulated totals, last receipt number, and — for petrol stations — the station structure), after which receipt numbering resumes from the last id known to the NRA and the device returns to `NORMAL`. The endpoints that perform it are not part of the published client contract; a client detecting `NEEDS_INIT` should surface it and route the device to service.

Device SIM mode and replacing the SIM card

`/system/info` reports a `simMode` field describing whether the SIM in the device still matches the one bound to the registration at the NRA:

- `OK` — the modem's SIM (IMSI) matches the registration; the device operates normally.
- `SWAPPED` — the SIM was replaced. Detected at startup when the modem's IMSI differs from the stored one. The device prints a "SIM card changed — registration update required" notice and **locks**: every request is rejected with **HTTP 423** except `POST /change-registration` and the read-only status endpoints `GET /system/info`, `/hello`, `/registration`, `/nra-registration`, `/system/diagnostic`. Reports are blocked too.

To recover, call `POST /change-registration` with the new **MSISDN** (and any other changed fields). The new SIM's IMSI is read from the modem automatically and sent to the NRA, so you do not provide it. On success the device returns to `OK`.

Device GSM connectivity mode

`/system/info` reports a `gsmMode` field describing whether the modem is still connected to the operator network:

- `OK` — connectivity is healthy; the device operates normally.
- `NO_CONNECTION` — after every Z-report (real or virtual) the device verifies operator-network registration. Three consecutive failures (the read-only `gsm_max_z_failures` limit, fixed at `3`) put the device into a **read-only** lock: fiscal/write operations return **HTTP 423** (`status 2330`) while reports and exports stay available. The failure counter is persisted, so the lock survives a restart.

Recovery is automatic and needs no client action: any successful NRA communication clears the lock, and while locked the service's maintenance loop re-tests the link every 30s and unlocks once the operator connection returns. An automatic status bon is printed on both the lock and the unlock.

Emergency mode (overriding the NRA-connection blocks)

Two of the locks above are caused by the device being unable to reach the NRA rather than by a data or registration problem: the **overdue-unsent-receipts** lock (when receipts cannot be sent for longer than the read-only `unsent_receipts_max_age_hours` limit, fixed at 72h) and the **GSM** `NO_CONNECTION` lock. An operator can authorize an **emergency mode** that keeps the device issuing receipts while one of these connection blocks is active:

- `POST /system/enable-emergency-mode` — body `{ "deviceKey": "..." }`. Enables the override.
- `POST /system/disable-emergency-mode` — body `{ "deviceKey": "..." }`. Returns to normal blocking.
- `GET /system/emergency-mode` — reads the current state (no `deviceKey` needed).

Both POST bodies optionally carry the authorizing **operator** and free-text **comments**, printed on the status bon. Supply `operator: { id, name }` **or** `operatorId` (an existing operator), but not both — the operator is printed in the receipt header exactly like on every other bon. `comments` is an array of COMMENT items in the same `{ left, center, right }` shape used by sell/storno/print-nf receipts, printed in the body after the status line. A bad operator returns **HTTP 422** (`status 2336`); malformed `comments` return **HTTP 422** (`status 2337`).

The `deviceKey` must equal the **update-server registration key** (the `registrationKey` stored at registration with the update server). A missing key returns **HTTP 422** (`status 2331`), a wrong key **HTTP 403** (`status 2332`). If the device is already in the requested state the call is a no-op: **HTTP 409** with `status 2333` (already on) or `2334` (already off), and no bon is printed.

Emergency mode overrides **only** the NRA-connection blocks. The integrity/registration locks — a foreign or uninitialized KLEN (`klenMode`), a swapped SIM (`simMode`) and a deregistered device — are **never** overridden. The state is persisted, so it survives a restart until explicitly disabled. Enabling and disabling each print a status bon; after disabling, if there is still no NRA connection, receipt issuance is blocked again.

`/system/info` (and the `deviceState` block in `/system/diagnostic`, also printed on the diagnostic bon) reports the emergency state and the consolidated blocking status:

- `emergencyMode` — `ON` / `OFF`.
- `writesBlocked` — `true` when fiscal/write operations are currently rejected (HTTP 423), accounting for the emergency override.
- `blockReason` — which lock is active: `NONE`, `KLEN_FOREIGN`, `KLEN_NEEDS_INIT`, `SIM_SWAPPED`, `DEREGISTERED`, or `GSM_NO_CONNECTION`. While emergency mode is `ON`, a GSM `NO_CONNECTION` lock yields `NONE` here (the override suppresses it).

Normal daily-use surface

These endpoints are the main operational surface for ordinary fiscal work:

- `/sell`
- `/storno`
- `/duplicate`
- `/cashInOut`
- `/report/X_REPORT`
- `/report/Z_REPORT` when allowed by fiscal state
- `/journal` for readback, reconciliation, and confirmation

- `/exports/klen` for downloadable KLEN/journal data without printing a journal report receipt
- `/virtual/render-receipt/{id}` to render a single virtual receipt as plain text

Integrator note:

- this is the surface most sales software will use regularly
- device state, printer availability, VAT state, and registration state still matter even for this daily-use group
- `/exports/klen` is a read/export path; use `/journal` when the workflow intentionally requires journal report receipt behavior
- `/virtual/render-receipt/{id}` returns the same formatted text as `/exports/klen?resultFormat=formatted_text` for one journal id, inline (not as a download), and only for virtual receipts (non-virtual entries return `409`, missing ids return `404`)

Support and device-management surface

These endpoints are not part of the daily cashier flow but belong to the published contract:

- `GET /system/diagnostic` — full device snapshot, also printed as a diagnostic bon
- `GET /exports/fiscal-memory`, `GET /exports/klen`
- `GET /modem-apn`, `GET /system/modem-info`, `GET/PUT /system/network-config`
- `GET /system/reboot`

Integrator note:

- expose them to operator/admin or device-management tooling rather than to the main cashier flow
- the device additionally implements diagnostic, lab and machine-to-machine endpoints (raw NRA traffic, modem control, log access, `nra_client` transport). They are intentionally omitted from this documentation and from the published OpenAPI spec — they are support tooling, not an integration contract, and may change without notice

API Groups In Practice

`nra`

The `nra` group exists to implement the device-side logic required for communication with the NRA server.

It is best understood as several related subgroups:

Registration-changing endpoints

- `/registration`
- `/change-registration`
- `/deactivate-registration`

These endpoints accept request data, validate the local preconditions, and then attempt to update the NRA server.

Operational behavior:

- If the NRA-side operation succeeds, the service updates the local database state and prints the corresponding receipt/document.
- If communication with the NRA server fails, execution does not continue into the local state-change path.
- These endpoints are therefore not just local CRUD operations; they are synchronized NRA lifecycle operations.

Petrol stations (`FDType` 3 and 31) both send a `petrol_station` object here, but the two types differ:

- `FDType` 31 is a petrol station **with** a level-gauge system. Its full layout (terminals, dispensers, nozzles, `GNo`) is reported to NRA, and the gauge/counter/delivery endpoints (`/petrol-station/gauge-state*`, `/petrol-station/delivery-*`) are available.

- `FDType` 3 is a petrol station **without** a level gauge. Its `petrol_station` object is used only to define the fuels sold; the level-gauge/counter data (tank levels, `dispensers`, `nozzles`, `GNo`) is silently dropped and never sent to NRA. The fuels reach NRA only as sales inside the X/Z reports, fuel changes go through `/change-registration`, and the gauge/counter/delivery endpoints reject the device with `409`.

Registration read, sync, and recover endpoints

- `/nra-registration`
- `/nra-registration-sync`
- `/nra-registration-recover`

These endpoints are used when the client needs live NRA-side registration data or needs to reconcile the device state with NRA.

Operational behavior:

- `/nra-registration` fetches registration data directly from the NRA server and returns it without updating the local stored registration.
- `/nra-registration-sync` fetches the current NRA registration data and attempts to overwrite/update the local stored registration with it.
- `/nra-registration-recover` is used when device-side data is missing; it uses request-supplied identification data to retrieve the NRA-side state and then attempts to rebuild/update the local registration data.

NRA update tasks

The device keeps periodic tasks describing when data must be sent to the NRA server. They may be created during registration, updated later according to NRA protocol, or arrive via SMS, which the background NRA program reads and applies to the local task state.

Integrator note:

- this is internal scheduling state; the endpoints that expose, transport or force it are not part of the published contract
- what a client needs to observe — registration state, connectivity, blocking status — is reported by `GET /system/info`

`system`

The `system` group exposes device status, diagnostics, emergency mode, date/time and network configuration.

Read-only connectivity endpoints:

- `GET /modem-apn` reads the currently active APN from the modem and returns `{"apn": "<value>"}`.
- `GET /system/modem-info` returns a modem status snapshot (operator registration, signal, IMSI-related state).
- `GET/PUT /system/network-config` reads and updates the persistent Ethernet startup configuration.

Integrator note:

- modem power control, APN switching and the raw NRA protocol pass-through are diagnostic endpoints outside the published contract; they exist for support tooling and must not be integrated against.

Canonical Reference

- Use openapi.md and /resources/api as the source of truth for endpoint contracts.
- Use this page only for orientation; keep contract-level truth in Swagger.

Deployment And Trust Boundary Assumptions

The device API is exposed over HTTP and is intended to be reachable from the local deployment environment in which the fiscal device operates.

Important assumptions:

- the API does not provide built-in token-based authentication
- the API does not provide a general authorization model with user roles or access tokens
- callers are expected to run within a trusted deployment/network environment around the device

Integrator note:

- if an installation requires stronger access control, that protection must be provided by the surrounding deployment architecture, not by the device API itself

Current Documentation Note

The historical Markdown endpoint list was broader but not always synchronized with the real route registration. The OpenAPI spec is now the canonical contract artifact and should be reconciled against runtime behavior first.

Related Docs

- constraints-and-behavior.md
- exports.md
- integration-flows.md
- openapi.md

Owner: Backend/Embedded team Last updated: 2026-05-19

Integration Flows

Purpose

Describe the main integration paths at a workflow level without duplicating the detailed API contract.

Scope

Common lifecycle flows and the documents that consumers should consult for exact request and response definitions.

Audience

- Developers implementing client-side sequencing and error handling

Typical Flows

1. Onboarding and initial setup flow

This is the standard path for an integrator starting the integration of a device that has already been provisioned in service or factory conditions.

Recommended sequence:

1. Check that the service is reachable and healthy
   - typical entrypoints: `/hello`, `/health`
2. Read the current local configuration/state that matters for setup
   - typical entrypoints: `/config`, `/header`, `/vat`, `/payment/labels`
3. Apply customer-specific local setup
   - configure company/header data through `/header`
   - configure VAT through `/vat`
   - configure device settings through `/config`
   - configure payment labels through `/payment/labels`
   - set date/time through the date-time endpoint
4. Verify the device is ready for registration and fiscal work

Important data checkpoints:

- `/hello` and `/health` should confirm that the service is reachable before configuration starts
- header/company data must match the customer setup
- VAT must be configured before normal reporting/fiscal work
- date/time should be set correctly before registration and normal daily operation
- configuration writes should be followed by a read-back check when the integrator needs to verify the effective stored state
- setup operations are state-dependent and should be completed before the device enters restricted fiscal states

2. NRA registration lifecycle flow

After local setup, the device is typically moved into the NRA registration lifecycle.

Recommended sequence:

1. Start initial registration with `/registration`
2. If the device already has registration data and requires a synchronized change, use `/change-registration`

3. If the device must be deactivated through the NRA flow, use `/deactivate-registration`
4. If live NRA-side registration data must be inspected without local overwrite, use `/nra-registration`
5. If local registration must be synchronized from NRA, use `/nra-registration-sync`
6. If device-side registration data is missing and must be rebuilt from NRA-side information, use `/nra-registration-recover`

Important data checkpoints:

- a successful `/registration` or `/change-registration` call should be treated as both an NRA-side and local-state transition
- after registration-changing operations, the integrator should verify the resulting stored state with `/registration`
- registration-changing flows are synchronized with NRA, not just local updates
- on successful NRA update, the service updates local state and produces the corresponding local receipt/document
- if NRA communication fails, the local state-change path does not continue as if the operation had succeeded
- `/nra-registration` is a live NRA read path, not a local-state update
- `/nra-registration-sync` is a local overwrite/synchronization path based on NRA-side data
- `/nra-registration-recover` should be used only when device-side data is missing or incomplete enough that recovery is required

3. Daily fiscal operations flow

Once the device is configured and registered, the integrator can drive the daily fiscal workflow.

Typical sequence:

1. Send a sale request with `/sell`
2. If needed, perform a storno with `/storno`
3. If needed, request a duplicate with `/duplicate`
4. If needed, perform a cash operation with `/cashInOut`
5. Read the journal when confirmation, reconciliation, or audit visibility is needed

Important data checkpoints:

- a successful sale-like operation should return a success response and create/update the expected local journal/history state
- if the integrator needs confirmation after a document-producing call, `/journal` is the natural follow-up read path
- receipt-producing requests that support operator attribution may send either `operator: { id, name }` or `operatorId`, but not both
- `operator` creates or renames the local operator record where supported; `operatorId` must already exist and is expanded into the stored receipt data
- omitting both operator fields preserves the legacy behavior
- receipt-producing operations are state-dependent
- application-level `status` values must be checked together with HTTP result codes
- successful operations affect local journal/history and fiscal state
- if printer/fiscal execution fails, the operation should be treated as failed rather than partially accepted

4. Reporting flow

Reports are typically used after the device is already configured and in operational use.

Typical sequence:

1. Use `/report/X_REPORT` for operational/day-state visibility
2. Use `/report/Z_REPORT` for fiscal closing/reporting flow when allowed by device state
3. Use journal and report-read endpoints when historical confirmation is needed

Important data checkpoints:

- the integrator should not treat `/report/Z_REPORT` as a harmless read; it changes report/fiscal state
- before requesting a Z report, the integrator should assume that registration state, VAT state, and fiscal-memory capacity may be validated by the backend
- GET-style printed report endpoints that accept operator attribution use `operatorId` in the query string
- reporting depends on current registration, VAT, fiscal-memory, and report-state conditions
- a Z report is not just a read operation; it changes fiscal/report state

5. Firmware and update behavior

Firmware updates are **not** an integration responsibility. The device checks the update server on its own schedule, downloads and prepares an update at runtime, and the boot-time installer applies it during the next startup. The endpoints that drive and inspect this belong to device-management tooling and are not part of the published client contract.

What a client should know:

- an update is prepared at runtime but installed only on restart, so the running version can lag behind a "downloaded" update
- `GET /system/info` reports the version actually running
- update failures surface as `1400-1499` status codes and are update-lifecycle problems, not fiscal errors

6. Maintenance and support flow

Typical maintenance usage from an integration:

- `GET /system/info` for device, printer, registration and connectivity state
- `GET /system/diagnostic` when a full device snapshot (and its printed bon) is needed
- `GET /exports/fiscal-memory` and `GET /exports/klen` for backup, reconciliation and audit
- `GET /system/reboot` when the device must be restarted after a configuration change

Important data checkpoints:

- these are investigation and housekeeping paths, not part of the regular sales loop
- deeper troubleshooting (raw NRA traffic, protocol pass-through, modem and log access) is handled through diagnostic endpoints that are not part of the published contract; contact support rather than integrating against them

Where To Look For Details

- Local development environment: local-simulation.md
- Endpoint shapes: openapi.md
- Error handling: errors-and-status-codes.md
- Runtime caveats: constraints-and-behavior.md

Owner: Backend/Embedded team Last updated: 2026-03-31

Constraints and Behavior

Purpose

Describe integration-relevant behavior that is important to clients but is not part of the endpoint-by-endpoint reference.

Scope

Contract ownership, expected error formats, environment assumptions, and known documentation limitations.

Audience

- API consumers building reliable clients
- QA engineers validating expected behavior

Contract Rules

- `doc/swagger.yml` is the source of truth for endpoint paths, methods, and schema details.
- HTTP status codes communicate transport-level outcome and broad error class.
- JSON `status` values communicate application-specific outcome semantics where the backend uses them.

Integration Expectations

- Do not assume all endpoints share one perfectly normalized response shape.
- Validate against the specific endpoint schema in Swagger rather than against historical examples from old Markdown docs.
- Be careful with legacy or internal endpoints that exist for device workflows or maintenance tasks.

Important Runtime Constraints

The device API is state-dependent. A valid request shape is not enough by itself; many operations also depend on the current fiscal, registration, and printer state.

VAT and fiscal-state restrictions

- VAT configuration cannot be changed when there is an active Z-report state.
- Some setup/configuration operations are therefore valid only before the device enters normal fiscal operation or while the fiscal state permits them.

Report restrictions

- A Z report cannot be generated if there is no active registration.
- A Z report cannot be generated if there is no remaining space in fiscal memory.
- If VAT is not configured, report-related operations are not allowed.

Registration-dependent fiscal behavior

The device distinguishes three registration situations, and they behave differently:

- **Never registered** (`NOT_REGISTERED`): fiscal operations are still allowed to execute. They are treated as **non-fiscal** — printed and journaled to the KLEN, but **not** written into fiscal memory.

- **Registered** (`REGISTERED`): normal operation; fiscal documents are written to fiscal memory.
- **Deregistered** (`DEACTIVATED`): the device is **read-only** — see *Deregistration restrictions* below.

A first registration is refused while the KLEN already holds fiscal activity (any journal entry beyond configuration and cliché). `POST /registration` then returns **HTTP 409** (`status 2321`) and the KLEN must be reset before registering. Re-registration after a deregistration does not require a reset (no fiscal activity can accumulate in the read-only state between the two).

KLEN mode restrictions (foreign or blank KLEN)

- `/system/info` exposes a `klenMode` field: `NORMAL` , `READONLY` , or `NEEDS_INIT` .
- When the attached KLEN (local database) does not match the device's fiscal memory, the device enters `READONLY` (a foreign KLEN with data) or `NEEDS_INIT` (a fresh/blank KLEN). In both cases fiscal write operations are rejected with **HTTP 423** and reports are not journaled (numbers do not advance).
- In `NEEDS_INIT` the device prints a "new KLEN — initialization required" notice at startup. Returning it to `NORMAL` requires KLEN initialization, which is a service operation performed with endpoints outside the published client contract.

SIM-card swap restrictions

- `/system/info` exposes a `simMode` field: `OK` or `SWAPPED` .
- The SIM is bound to the registration at the NRA (its IMSI is stored and sent during registration). If the SIM is replaced, at the next startup the modem reports a different IMSI and the device enters `SWAPPED` , printing a "SIM card changed — registration update required" notice.
- In `SWAPPED` the device is **locked**: every request is rejected with **HTTP 423** *except* `POST /change-registration` and read-only status endpoints (`GET /system/info` , `/hello` , `/registration` , `/nra-registration` , `/system/diagnostic`). Unlike KLEN read-only mode, **reports are blocked too**.
- To recover, call `POST /change-registration` with the new **MSISDN** (phone number) and any other changed fields. The new SIM's IMSI is read from the modem automatically. On success the device returns to `OK` and resumes normal operation.

Deregistration restrictions (deactivated registration)

- After a successful `POST /deactivate-registration` the registration state becomes `DEACTIVATED` and the device is **read-only — exactly like a foreign KLEN**. Every fiscal/write operation (`/sell` , `/storno` , `/cashInOut` , `/report/X_REPORT` , `/report/Z_REPORT` , `/vat` , `/system/currency/apply-pending`) is rejected with **HTTP 423** (`status 2320`). Read-only reports and exports remain available (`GET /report/operators|articles|contracts` , `GET /report/fiscal-memory` , `GET /exports/klen` , `GET /journal`) and are not journaled.
- The recovery path depends on the **deactivation reason** (the `RCFD` value, 1–10, sent to `/deactivate-registration`):
  - **Reasons 1–8** — "fiscal memory removal" (overflow, owner change, termination, scrapping, FM damage, FM block error, putting-in-operation error, testing completed): the device is scrapped. There is **no recovery** — both `POST /registration` (423) and `POST /change-registration` (409) are refused.
  - **Reason 9** — temporary suspension of activity: re-activate with `POST /change-registration` . On success the device returns to `REGISTERED` .
  - **Reason 10** — other: re-register with the current data via `POST /registration` . On success the device returns to `REGISTERED` .
- This distinguishes a **never-registered** device (sales allowed as non-fiscal) from a **deregistered** one (read-only, only reports/exports).

GSM operator-connectivity restrictions

- `/system/info` exposes a `gsmMode` field: `OK` or `NO_CONNECTION` .
- After **every Z-report** (real *and* virtual) the device checks whether the modem is still registered on the operator network (`creg_status`). Each consecutive failed check — including an unreadable modem — increments a persisted counter; a connected check resets it to `0` .

- After **3 consecutive** failures (configurable via `gsm_max_z_failures`, default `3`) the device enters `NO_CONNECTION` and is **read-only — exactly like a foreign KLEN**: fiscal/write operations are rejected with **HTTP 423** (`status 2330`) while reports and exports remain available. The counter survives a restart, so a device that was locked stays locked until connectivity returns.
- An automatic "GSM operator connection" status bon is printed when the lock engages (explaining the block) and again when it clears (connection restored).
- Recovery is automatic and happens on either of:
  - **any successful NRA communication** (the `nra_client` posting a result back) — a reply proves the GSM link works; or
  - the **periodic connectivity recheck** driven by the service's maintenance loop (every 30s while locked), for when there is no traffic to send.

Printer availability restrictions

- If the printer cannot print, receipt/document-producing operations fail.
- In practice this means that sales, storno, duplicate, and other print-dependent receipt flows should be expected to fail when the printer is unavailable.

Virtual-mode behavior

- When the `virtual` mode/query flag is used, the system treats the printer as if it can print.
- This is useful for development, testing, and flows where physical print availability must be bypassed intentionally.

State-Sensitive Endpoint Families

Clients should group endpoint behavior by device state, not only by URL shape.

Setup and configuration endpoints

- Setup-oriented endpoints such as `/header`, `/config`, `/payment/labels`, and `/vat` should be treated as controlled configuration operations, not as arbitrary CRUD that can always run during active fiscal work.
- `/vat` is the clearest example: it cannot be changed while there is an active Z-report state.
- In practice, integrators should perform configuration writes during setup or during controlled maintenance windows when the fiscal state allows them.

Registration-changing endpoints

- `/registration`, `/change-registration`, and `/deactivate-registration` are synchronized device-and-NRA operations, not local-only data updates.
- These endpoints depend on valid local preconditions, valid request data, and successful communication with the NRA side.
- If the NRA-side step fails, the operation does not continue into the local commit path.
- Integrators should therefore treat registration-changing calls as externally dependent lifecycle operations rather than as always-available local settings writes.
- After a deregistration these endpoints are gated by the deactivation reason (see *Deregistration restrictions*): only reason 9 keeps `/change-registration` reachable, only reason 10 keeps `/registration` reachable, and reasons 1–8 keep the device permanently locked.

Registration read, sync, and recover endpoints

- `/nra-registration` is a read-from-NRA operation and should be treated as a live external lookup.
- `/nra-registration-sync` and `/nra-registration-recover` are reconciliation operations that can affect local registration state.
- These endpoints are useful when device state and NRA state must be compared or rebuilt, not as part of every normal cashier workflow.

Fiscal-operation endpoints

- Receipt-producing endpoints such as `/sell`, `/storno`, `/duplicate`, and `/cashInOut` depend on printer availability unless virtual mode is used intentionally.
- Registration state changes the fiscal meaning of these operations: when the device is not registered, they may still execute, but they are treated as non-fiscal and are not written to fiscal memory.
- Clients should therefore not assume that a syntactically valid sale request always produces a normal registered fiscal record.
- Where supported, receipt-producing endpoints may accept operator attribution as either `operator: { id, name }` or `operatorId`, but not both.
- `operatorId` is a lookup reference to an existing local operator. A missing operator id is a bad request, while omitting both operator fields preserves legacy behavior.
- Operator attribution is stored in the journal receipt payload so the receipt can be recreated later.

Report endpoints

- Report operations are sensitive to both registration and fiscal state.
- `/report/Z_REPORT` requires an active registration and available space in fiscal memory.
- Report flows also depend on VAT configuration being present.
- Printed GET report endpoints that support operator attribution use `operatorId` as a query parameter.
- Clients should treat report endpoints as state-gated operational steps, especially when automating end-of-day behavior.

NRA task endpoints

- NRA synchronization tasks are internal workflow state. The endpoints that expose, transport and force them (used by `nra_client` and by support tooling) are deliberately absent from this documentation and from the published OpenAPI spec, and integrations must not depend on them.
- Client-visible consequences of that workflow are reported through `GET /system/info` and the registration endpoints.

Streaming Responses

Some endpoints return HTTP chunked streaming responses using `Transfer-Encoding: chunked`. The service does not accumulate the full result in memory before responding; it fetches data from the database in chunks and writes each chunk to the response as it is produced.

Endpoints that stream

- `GET /journal` — all output formats (JSON, `formatted_text`, CSV) are streamed. The response body is fully valid JSON/text/CSV when the transfer completes; there is no partial-response indicator in the content itself.
- `GET /exports/klen` — JSON, CSV, and formatted-text formats are streamed. Binary (`resultFormat=binary`) is not streamed in the same sense; it is sent as a single continuous file read.

Client implications

- Clients must consume the full response body before treating it as complete. Do not parse the body until the HTTP transfer finishes.
- `Content-Length` is not set on streaming responses. Clients that require `Content-Length` before consuming cannot use these endpoints directly without a buffering proxy.
- JSON streaming responses are complete, valid JSON arrays once the transfer ends. There are no NDJSON or partial-object conventions.

Journal print chunking

When `GET /journal` is requested with `limit > 200`, the optional printed journal report receipt is produced in chunks of 200 entries each rather than as a single print command set. This is transparent to the caller; the HTTP response is

unchanged.

Known Caveats

- The retired `doc/REST_ENDPOINTS.md` overview did not fully match actual route registration in `src/rest/http_handlers.c` and the `rest_*_handlers.c` files.
- Some path names historically documented in Markdown differ from the code. Example mismatches already observed:
  - Markdown used `/company`, while code registers `/header`
  - Markdown used `/payment-labels`, while code registers `/payment/labels`
- External-facing landing pages should be treated as navigational and explanatory, not as an alternative contract source.

Legacy And Compatibility Notes

- Historical Markdown docs may refer to endpoint names that are no longer the canonical route names. Clients should follow Swagger and runtime behavior instead of old prose examples.
- Not all endpoint families behave like normalized local CRUD:
  - NRA-facing endpoints may depend on live communication with external NRA services
  - some NRA-related responses may be closer to passthrough or protocol-oriented behavior than to the usual local wrapper style
- Older examples may assume that registration is a hard prerequisite for every sale-like operation. The actual behavior is more nuanced: some operations can still run in non-fiscal mode when the device is not registered.
- Printer availability and virtual mode also affect behavior in ways that are easy to miss if a client was built only from historical examples or happy-path manual testing.

Related Docs

- errors-and-status-codes.md
- openapi.md

Owner: Backend/Embedded team Last updated: 2026-06-07

Errors and Status Codes

Purpose

Explain the two-layer error model used by the service so API consumers know what to inspect in client code.

Scope

HTTP statuses, application `status` values, and where to find the exact mapping.

Audience

- Client developers
- QA and support engineers investigating API failures

Error Model

- HTTP status indicates transport-level success or failure and the broad class of the result.
- JSON `status` indicates application-specific semantics where the endpoint returns the standard wrapper.
- Exact `status` code meanings are documented in the schemas and endpoint descriptions in the OpenAPI contract exposed through the runtime Swagger/OpenAPI reference at /resources/api.

Client Guidance

- Always inspect both HTTP status and response body.
- Do not assume that every endpoint returns the same success or error envelope.
- Prefer endpoint-specific Swagger schemas over old generic examples.

Status Code Families

The JSON `status` values are grouped by subsystem/domain. This is useful for client-side error handling because the numeric range already tells you what kind of failure you are dealing with.

Important families:

- `0` : success
- `1` : generic not-implemented marker
- `1000-1099` : generic/common request and persistence failures
- `1100-1199` : sales/document creation
- `1200-1299` : reports
- `1300-1399` : NRA registration flow
- `1400-1499` : firmware/update lifecycle
- `1500-1599` : setup entities
- `1600-1699` : journal and fiscal memory
- `1700-1799` : exports
- `1800-1899` : petrol-station flows
- `1900-1999` : system/provisioning
- `2000-2099` : USN
- `2100-2199` : NRA communication and NRA update tasks
- `2200-2299` : date-time, invoice-range, and file-serving related endpoints

- `2300-2399` : device-mode locks — foreign/blank KLEN (`2301-2305`), SIM swap (`2310`), deregistration (`2320` deregistered/read-only, `2321` KLEN not empty for a first registration), and GSM operator connectivity (`2330` no connection/read-only after consecutive failed post-Z checks). These accompany **HTTP 423** (operation blocked) or **HTTP 409** (`2321`).

For the exact mapping, use the OpenAPI contract exposed through the runtime Swagger/OpenAPI reference at /resources/api.

NRA Passthrough Versus Local Wrapper Errors

Not all NRA-related endpoints behave the same way when NRA returns a structured payload.

Local wrapper error model

Many endpoints use the standard local JSON wrapper with:

- HTTP status
- JSON `status`
- local error message

This is the normal pattern for local validation failures, local persistence failures, and many backend-side runtime failures.

NRA passthrough behavior

Some NRA-related endpoints may return the NRA payload itself instead of the standard local wrapper.

Important patterns:

- `/nra-registration` may return `200` and pass through a valid NRA payload even when the NRA-side `Status` is non-zero
- `/nra-registration-recover` may also return `200` and pass through a valid NRA payload with non-zero NRA `Status`
- `/nra-registration-sync` is different: if NRA returns a valid payload with non-zero `Status` , the endpoint returns `409` and does not update local state
- `/registration` , `/change-registration` , and `/deactivate-registration` may also return structured NRA rejection payloads instead of a normal local success path when NRA rejects an otherwise valid request

Integrator rule:

- for NRA-related endpoints, do not assume that every error-like condition is represented only by the local JSON wrapper
- always validate the actual endpoint response shape against Swagger

Practical Error Examples

Validation/setup example

Example situation:

- the client attempts a setup or sales-related operation with invalid or incomplete request data

Typical result:

- HTTP `400` or `422`
- local JSON error wrapper
- JSON `status` in the generic/setup/sales range depending on the endpoint

Typical client action:

- treat this as a client-side/request-side problem
- fix the request body or the local device state before retrying

NRA communication example

Example situation:

- a registration-changing endpoint reaches NRA transport failure or receives an NRA-side rejection

Typical result:

- transport-level NRA failure often appears as HTTP `502`
- NRA rejection may appear as a structured NRA payload, depending on the endpoint
- local state should not be treated as updated unless the endpoint semantics explicitly indicate success

Typical client action:

- distinguish network/upstream failure from local validation failure
- do not assume the registration was changed locally just because a request was sent

Firmware/update example

Example situation:

- the client asks the device to check/download/install an update, but the update is not ready or cannot be resolved

Typical result:

- local wrapper error with JSON `status` in the `1400-1499` range
- examples include:
  - update-server registration key missing
  - update check failed
  - requested update ID invalid or not found
  - firmware package not downloaded yet

Typical client action:

- treat firmware/update failures as update-lifecycle problems, not as sales/runtime fiscal errors
- distinguish "no prepared update" from "prepared update available" and from "installer must apply later at startup"

Where To Get Concrete Payload Examples

- For exact success and error payload shapes, use the endpoint-specific schemas and examples in the OpenAPI contract exposed through the runtime Swagger/OpenAPI reference at /resources/api.
- Prefer those endpoint-level definitions over generic Markdown snippets, because several endpoint families use different wrappers or passthrough behavior.
- This is especially important for NRA-related endpoints, where the response shape may differ from the usual local JSON wrapper.

Related Docs

- openapi.md
- constraints-and-behavior.md

Owner: Backend/Embedded team Last updated: 2026-06-07

Service Configuration

Purpose

Provide a public-facing reference for the service configuration surface, including what each variable means, where it comes from, and whether it can be changed through the API.

Scope

This page covers:

- effective configuration returned by `GET /config`
- fields accepted by `POST /config` and `PUT /config`
- persistent Ethernet startup configuration exposed through `GET /system/network-config` and `PUT /system/network-config`
- startup inputs that shape runtime behavior (JSON + environment)

This page complements Swagger. Swagger remains the schema contract source of truth, while this document explains operational meaning and runtime behavior.

Audience

- Integrators who need to understand available configuration fields
- QA engineers preparing setup and regression scenarios
- Support and deployment engineers comparing startup settings with effective runtime state

Source Basis

- `doc/swagger.yml`
- `src/rest/rest_config_handlers.c`
- `src/entities/config.c`
- `src/utils/init_helper.c`
- `src/main.c`
- `system-scripts/aclas/default-settings.json`
- `system-scripts/aclas/nsys_env.sh`

How Configuration Works

The service combines configuration from two places:

1. Startup inputs loaded when the service starts (JSON file plus environment variables, with env taking precedence where implemented).
2. Persisted configuration managed through `/config`.
3. Persistent Ethernet startup configuration managed through `/system/network-config`.

`GET /config` returns the effective configuration. It includes both persisted fields and read-only fields injected from startup settings.

`POST /config` and `PUT /config` only persist writable fields. Read-only startup fields are ignored on write, even if they are sent in the request.

`GET /system/network-config` and `PUT /system/network-config` operate on the boot-time Ethernet config file used by the ACLAS startup scripts. This is separate from `/config`.

The response always includes:

- `global_settings_update_interval_sec`
- `max_fm_blocks_warning_threshold`

If these values are missing from persisted state, the service injects defaults before returning `/config`.

Persistent Ethernet Startup Configuration

`/system/network-config` is the public API for the Ethernet startup mode used by the ACLAS boot scripts.

It is intentionally separate from `/config` because it controls pre-service boot behavior rather than normal runtime service configuration.

What it controls

The persistent file-backed network configuration drives the Ethernet part of the boot flow in `app.sh` and `setup_eth.sh`.

Supported modes are:

Field	Type	Description	Notes
`network_mode`	string	Ethernet startup mode.	Allowed values: `default_static`, `dhcp`, `custom_static`.
`eth_device`	string	Ethernet interface name.	Usually `eth0`.
`static_ip`	string	IPv4 address for static modes.	Used by `default_static` and `custom_static`.
`netmask`	string	IPv4 netmask for static modes.	Used by `default_static` and `custom_static`.
`gateway`	string	Optional IPv4 default gateway.	Empty string means no default gateway is configured.

Mode behavior

Mode	Behavior
`default_static`	Uses the built-in fallback ACLAS address `192.168.1.87/255.255.0.0`.
`dhcp`	First applies `192.168.1.87/255.255.0.0`, then starts DHCP for the configured Ethernet interface so a lease can replace the fallback address.
`custom_static`	Uses the provided `static_ip`, `netmask`, and optional `gateway`.

Read/write behavior

- `GET /system/network-config` returns the effective persistent startup configuration.
- If the file does not exist yet, the endpoint returns the built-in default configuration.
- `PUT /system/network-config` validates and saves the config atomically to the persistent network config file.
- The response includes `pending_restart=true` after a successful write.

Apply behavior

`PUT /system/network-config` does not reconfigure the live interface immediately.

The saved config is applied on the next boot/startup cycle when the device runs:

- `app.sh`
- `setup_eth.sh`

To trigger an immediate reboot after saving, add the query parameter `rebootDevice=true`:

```
PUT /system/network-config?rebootDevice=true
```

The response is sent first, then the device reboots after ~2 seconds. This separation is intentional because changing the active IP at runtime can interrupt the same REST session that requested the change.

Startup printout

At the end of startup the device prints the short "printer ready" label, and under it the basic network configuration, so the operator can see how to reach the device without printing the full diagnostic:

Line	Source
Network mode	`network_mode` from the persistent network config file, as a display name (`DHCP` , `Static (default)` , `Static (custom)`).
Ethernet device	`eth_device` from the same file.
Ethernet IP	The address the interface actually runs with right now, read from the kernel. Under `dhcp` this is the leased (or fallback) address, not a value from the file.
GW	The interface's default-route gateway, read from the kernel.
Ethernet MAC	The interface hardware address.

A value the device cannot determine (no such interface, no default route) prints as `---` . When `print_full_diagnostic_on_boot` is enabled, the full diagnostic bon is printed at boot instead of this label; its network block carries the same mode, device, IP and MAC (plus the TCP port).

Effective `/config` Reference

Read-only fields injected from startup

These fields are visible in `GET /config` , but writes through `/config` do not change them.

Field	Type	Description	Notes
`port`	integer	HTTP port used by the service.	Comes from startup `service_port`. Returned as read-only.
`bind_address`	string	IPv4 interface the plaintext HTTP service binds to.	Absent/empty/`0.0.0.0` = all interfaces (default). Set to `127.0.0.1` to restrict the service to loopback — typical once `nsys-tls-proxy` fronts it on the LAN. **Caveat:** internal clients (`nra client`, `nra_cron`, auto-Z) reach the service via `http://localhost`, so the only safe values are all interfaces or `127.0.0.1`; binding to only an external IP breaks those loopback calls. Startup-only; a change requires a restart. Only present when configured.
`db_path`	string	Path to the SQLite database file.	Deployment-level setting. Returned as read-only.
`printer_type`	string	Printer implementation used by the service.	Swagger enum: `aclas`, `dummy`, `fiscat`. Comes from startup `printer_type`.
`fm_backend_type`	string	Fiscal memory backend type.	Swagger enum: `aclas`, `dummy`, `file`. Comes from startup `fiscal_memory_type`.
`printer_config`	object	Low-level printer transport settings.	Read-only block created from startup/device configuration.
`printer_config.device_path`	string	Printer device path.	Example: `/dev/ttyp0`.
`printer_config.baud_rate`	integer	Serial baud rate for the printer connection.	Example: `115200`.
`printer_config.data_bits`	integer	Number of serial data bits.	Example: `8`.
`printer_config.parity`	integer	Serial parity mode.	`0 = none`, `1 = odd`, `2 = even`.
`printer_config.stop_bits`	integer	Number of serial stop bits.	Example: `1`.
`printer_config.is_rtscts`	integer	RTS/CTS hardware flow control flag.	`0 = disabled`, `1 = enabled`.
`modem_status_script`	string	Path to the script used to collect modem status.	Used by diagnostics/device-info flows.
`modem_device_file`	string	Device file for the modem.	Example: `/dev/ttyUSB0`.
`eth_device`	string	Network interface name used by some diagnostics.	Example: `eth0`. Exposed as read-only in runtime config even though it is not currently documented in Swagger `Config`.
`sms_save_path`	string	Directory where incoming SMS messages are stored.	Also used by SMS export flows.
`nra_access_url`	string	NRA endpoint base URL used by the service.	Read-only startup connectivity setting.
`resource_path`	string	Directory for runtime resources such as localization files, the fixed fiscal-logo resource `bg_map.bmp`, and extracted update resources.	Returned as read-only. The footer logo is no longer configured through `/config`; the service always looks for `resource_path/bg_map.bmp`.
`documentation_path`	string	Directory for browser-facing runtime documentation assets such as `/resources/doc` and `/resources/api`.	Returned as read-only when configured. On ACLAS the default startup value is `/tmp/sd/nsys/docs`.
`unsent_receipts_max_age_hours`	integer	Maximum age in hours for unsent fiscal receipts (FISCAL_RECEIPT, STORNO_RECEIPT, INVOICE) before new receipt issuance is blocked.	Regulatory limit, fixed at its compiled default `72` (3 days). Seeded from startup config; writes through `/config` are stripped.
`gsm_max_z_failures`	integer	Consecutive post-Z-report GSM operator-connectivity failures before the device locks (read-only).	Regulatory limit, fixed at its compiled default `3`. A non-positive value disables the lock. Seeded from startup config; writes through `/config` are stripped.

Writable fields exposed through `/config`

These fields can be created or updated through `POST /config` and `PUT /config`.

Field	Type	Description	Notes
`ip_whitelist`	array of strings	Client-IP allow-list for incoming requests, HTTP and HTTPS alike.	When set to a non-empty array, only requests whose client address matches an entry are served; all others get `403` (`status` 2350). When absent or an empty array, the allow-list is disabled and every client is accepted. IPv4-mapped IPv6 clients are normalized to plain IPv4. A single comma/whitespace-separated string is also accepted. **Must include `127.0.0.1` when non-empty:** the internal callers (`nra_client`, `nra_cron`, auto-Z) reach the service over loopback and are subject to the same list. **HTTPS:** enforced by `nsys-tls-proxy` on the real client address before the TLS handshake, because the service sees only the proxy's loopback address for those connections — see the HTTPS note below.
`operator_label`	string	Label printed in the operator area of receipts.	Default from startup helper: `"----"`.
`footer_line1`	string	First custom footer line on receipts.	Printed near the receipt footer.
`footer_line2`	string	Second custom footer line on receipts.	Printed near the receipt footer.
`localization_language`	string	Active localization resource code.	Loads `resource_path/localization/<language>.json` on demand. Default: `"bg"`.
`receipt_config`	object	Receipt layout and print behavior settings.	Must be an object when provided.
`receipt_config.cut_type`	string	Paper cut mode after printing.	Swagger enum: `FULL`, `PARTIAL`, `NONE`. Default: `FULL`.
`receipt_config.delimiter_char`	string	Character used for delimiter lines.	When set to `-`, the service fills the line with that character.
`receipt_config.before_print_feed`	integer	Number of feed lines before receipt content starts.	Default: `3`.
`receipt_config.before_cut_feed`	integer	Number of feed lines before the cut command.	Default: `5`.
`receipt_config.encoding`	string	Printer character encoding.	Swagger enum: `CP437`, `KATAKANA`, `CP850`, `CP860`, `CP863`, `CP1251`, `CP866`. Default: `CP1251`.
`receipt_config.receipt_ext_format`	boolean	Enables extended receipt formatting.	When enabled, quantity, price, and total are printed on separate rows where supported.
`receipt_config.receipt_print_vat`	boolean	Controls whether VAT details are printed on receipts.	Similar to invoice-style VAT detail output.
`receipt_config.print_dark`	integer	Thermal print-head darkness/concentration (ACLAS).	`1` – `250` (higher = darker). Default `2`; out-of-range values fall back to the default. Applied at startup and after each config update.
`receipt_config.print_speed`	integer	Print speed index (ACLAS).	`1` – `4` (`1=160`, `2=180`, `3=200`, `4=230`; higher = faster). Default `2`; out-of-range values fall back to the default. Applied at startup and after each config update.
`receipt_config.header_style`	object	Print style for receipt headers.	Uses the `PrintStyle` schema from Swagger.
`receipt_config.body_style`	object	Print style for the receipt body.	Uses the `PrintStyle` schema from Swagger.
`receipt_config.footer_style`	object	Print style for footer content.	Uses the `PrintStyle` schema from Swagger.
`receipt_config.total_style`	object	Print style for totals.	Uses the `PrintStyle` schema from Swagger.
`auto_z_report_enabled`	boolean	Enables automatic previous-day Z report generation.	The service checks periodically and triggers after a new local day has started; the generated Z report is timestamped at `23:59:59` for the previous open day. Default: `false`.

Field	Type	Description	Notes
`allow_empty_z_report`	boolean	Allows generating Z reports even when there are no transactions.	Affects whether automatic Z generation proceeds when the day is empty. Default: `false`.
`auto_z_report_virtual`	boolean	Controls whether the automatic Z flow calls `POST /report/Z REPORT?virtual=true`.	Used only by the automatic flow. Default: `true`.
`auto_backup_enabled`	boolean	Enables automatic daily database backup generation.	Swagger documents this field and startup defaults set it to `true`.
`global_settings_update_interval_sec`	integer	Interval in seconds between attempts to refresh global settings from the update server.	Default: `1800` seconds (`30` minutes). Always present in `GET /config`.
`max_fm_blocks_warning_threshold`	integer	Remaining fiscal-memory-block threshold that triggers a warning status.	Must be a non-negative integer. Default: `100`. Always present in `GET /config`.
`update_config`	object	Update-over-APN and periodic-check settings.	Runtime-editable section (mirrors `receipt_config`). All keys optional.
`update_config.update_via_apn`	string	When to temporarily switch APN for an update check.	`never` / `on_failure` / `always`. Default: `on_failure`.
`update_config.update_apn`	string	APN to switch to for updates when the update server is unreachable on the current route.	Empty disables APN switching entirely.
`update_config.update_apn_route_mode`	string	udhcpc route mode applied while on the update APN.	`default` / `both`. Default: `default`.
`update_config.update_apn_server_url`	string	Update-server base URL used only while on the update APN.	Empty reuses the top-level `update_server_url`.
`update_config.update_check_interval_sec`	integer	Minimum interval between automatic update checks by `nsys-update-client`.	Default: `7200` (2 hours). `0` disables periodic checking; an update check can still be triggered by device-management tooling.

Startup JSON Parameters

Startup values are loaded by `load_main_params()` from the JSON file passed to the binary, but many path-like fields are resolved with this precedence:

1. Environment variable from `nsys_env.sh`
2. JSON value (if env is missing)

On ACLAS devices, `system-scripts/aclas/default-settings.json` is intentionally minimal, while path and deployment locations are provided by `system-scripts/aclas/nsys_env.sh`.

That means path fields are effectively device-managed and should be treated as read-only from API clients. Some are visible in `GET /config`, others are internal-only.

JSON-first startup keys (from `default-settings.json`)

Startup parameter	Description	Runtime mapping / behavior
`service_port`	HTTP port for the service.	Reflected as read-only `/config.port`.
`https_port`	HTTPS port advertised in UDP discovery responses. Optional; default `443`. Set to `0` to advertise no HTTPS endpoint.	Not a port the service binds — it is the port the separate `nsys-tls-proxy` terminates TLS on (`NSYS_TLS_LISTEN_PORT` in `app.sh`). Read only at startup for the discovery responder; not exposed in `/config`.
`discovery_port`	UDP port the discovery listener binds.	Optional; default `45678`. Not exposed in `/config`.
`bind_address`	IPv4 interface the HTTP service binds to. Optional; absent/empty/`0.0.0.0` = all interfaces (default). Env override: `NSYS_BIND_ADDRESS`.	Reflected as read-only `/config.bind_address` (only when set). Startup-only; requires a restart to change.
`db_path`	Path to the SQLite database.	Reflected as read-only `/config.db_path`.
`printer_type`	Printer implementation to use.	Reflected as read-only `/config.printer_type`.
`fiscal_memory_type`	Fiscal memory backend implementation.	Reflected as read-only `/config.fm_backend_type`.
`log_file`	Log target file name or special target such as `stderr`, `stdout`, or `-`.	Used by logging initialization; not exposed in `/config`.
`log_path`	Preferred log directory path.	Used as `logs_dir` if provided. Not exposed in `/config`.
`logs_dir`	Log directory path.	Fallback when `log_path` is missing. Used by log export endpoints. Not exposed in `/config`.
`log_level`	Logging verbosity.	Used during startup logging initialization; not exposed in `/config`.
`modem_device_file`	Modem device file path.	Reflected as read-only `/config.modem_device_file`.
`eth_device`	Ethernet device/interface name.	Reflected as read-only runtime config field.
`nra_access_url`	NRA communication URL.	Reflected as read-only `/config.nra_access_url`.

Env-first startup keys (typically device-managed via `nsys_env.sh`)

Env variable	JSON fallback key	Description	Runtime mapping / behavior
`NSYS_DB_PATH`	`db_path`	Path to the SQLite database.	Reflected as read-only `/config.db_path`.
`NSYS_FM_FILE_PATH`	`fm_file_path`	Path to fiscal memory file backend.	Used by file FM backend; not exposed in `/config` by default.
`NSYS_MODEM_STATUS_SCRIPT`	`modem_status_script`	Script used to collect modem/device status.	Reflected as read-only `/config.modem_status_script`.
`NSYS_NRA_ACCESS_SCRIPT`	`nra_access_script`	Wrapper script for NRA calls.	Used internally by NRA access helper; not exposed in `/config` by default.
`NSYS_SMS_SAVE_PATH`	`sms_save_path`	Directory where received SMS messages are stored.	Reflected as read-only `/config.sms_save_path`.
`NSYS_BIN_PATH`	`install_bin_dir`	Directory where installed firmware binaries are copied.	Used by firmware install/update flows; not exposed in `/config`.
`NSYS_LIB1_PATH`	`install_lib_dir`	Primary directory where installed libraries are copied.	Used by firmware install/update flows; not exposed in `/config`.
`NSYS_LIB2_PATH`	`install_lib2_dir`	Secondary runtime library directory.	Used by firmware install/update flows; not exposed in `/config`.
`NSYS_LIB2_FILES`	`install_lib2_files`	Extra runtime libraries expected in `NSYS_LIB2_PATH`.	Used by firmware installer update script execution; not exposed in `/config`.
`NSYS_SYSTEM_SCRIPTS_PATH`	`install_system_scripts_dir`	Directory where system scripts shipped with updates are installed.	Used by firmware install/update flows; not exposed in `/config`.
`NSYS_SYSTEM_TOOLS_PATH`	`install_system_tools_dir`	Directory where bundled maintenance tools are installed.	Used by deployment/runtime helper scripts; not exposed in `/config`.
`NSYS_MIGRATIONS_PATH`	`install_migrations_dir`	Directory containing DB migration scripts.	Used by migration/update flows; not exposed in `/config`.
`FILE_STORE_DIR`	`file_store_dir`	Directory where uploaded files and update packages are stored.	Used by upload/download/update flows; not exposed in `/config`.
`TEMP_EXTRACT_DIR`	`temp_extract_dir`	Temporary directory for extracting update archives and building exports.	Used by firmware/update/export flows; not exposed in `/config`.
`NSYS_RESOURCE_PATH`	`resource_path`	Directory for static/runtime resources.	Reflected as read-only `/config.resource path`; logo is read from `resource_path/bg_map.bmp`.
`NSYS_DOCUMENTATION_PATH`	`documentation_path`	Directory for runtime documentation assets served through `/resources/doc` and `/resources/api`.	Reflected as read-only runtime config when configured. On ACLAS the default value is `/tmp/sd/nsys/docs`.
`NSYS_BACKUP_DIR`	`backup_dir`	Root directory for automatic backups.	Used by daily DB backup and firmware rollback; not exposed in `/config`.
`NSYS_LOGS_PATH`	`logs_dir` (then `log_path`)	Directory for log files.	Used by logging init and log export flows; not exposed in `/config`.

Runtime Notes

IP allow-list over HTTPS

`ip_whitelist` is enforced at two points, against the same config value:

- **Plaintext HTTP** — by the service itself, on the connection it accepts. Denied requests get the usual `403` with `status` 2350. Unchanged behavior.
- **HTTPS** — by `nsys-tls-proxy`, on the address of the client it accepts, before the TLS handshake. The proxy relays to the service from `127.0.0.1`, so the service cannot see who the TLS client really is; the check has to happen one

hop earlier.

Practical consequences:

- A denied HTTPS client gets the TCP connection closed, not a `403` body. There is no TLS session yet to carry one, and refusing before the handshake is what keeps an unauthorized peer from costing the device a handshake.
- A non-empty list must contain `127.0.0.1`, otherwise the loopback hop from the proxy — and the internal callers `nra_client`, `nra_cron` and auto-Z — are denied by the service.
- The proxy reads the list from the KLEN on every accepted connection, so `POST` / `PUT /config` changes take effect immediately, with no restart of either process.
- The proxy needs the KLEN path to do this: `--settings <file>` (as `app.sh` passes it) or the `NSYS_DB_PATH` environment variable. With neither, it logs a warning at startup and serves every client IP, leaving enforcement to the service alone.
- If the KLEN cannot be read or the stored config is unparsable, the proxy fails open and allows the client — matching the service, where an unset or malformed `ip_whitelist` allows everyone.

Localization source

Localization labels are no longer managed through database CRUD. The active localization payload is loaded from resource JSON files under `resource_path/localization/`.

The fiscal receipt footer logo is also no longer configured through `/config`. The service always loads the fixed resource file `bg_map.bmp` from `resource_path`, and receipt generation fails if that file is missing.

At the moment the service initializes the built-in default resource `bg.json` and exposes it through `GET /localization`. Future localization work should extend the resource-reading surface, not reintroduce persistence for labels.

Read-only write behavior

If a client sends read-only startup fields to `POST /config` or `PUT /config`, the service strips them before persistence. They will still appear in later `GET /config` responses because they are re-injected from startup configuration.

This read-only reinjection is especially important when reusing an older database on a newer runtime. Deployment-owned paths such as `resource_path` and `documentation_path` are taken from current startup configuration rather than trusted from older persisted config rows.

`/system/network-config` is different: it is a dedicated writable surface for boot-time Ethernet configuration and persists directly to the startup config file rather than to the main runtime config persistence.

Validation details

- `receipt_config`, when present, must be a JSON object.
- `auto_backup_enabled`, when present, must be a boolean.
- `localization_language`, when present, must be a string.
- `max_fm_blocks_warning_threshold`, when present, must be a non-negative integer.

Defaults currently injected by the service

Startup helper code initializes new/default configuration with these values:

Field	Default
`operator_label`	`"----"`
`footer_line1`	`""`
`footer_line2`	`""`
`localization_language`	`"bg"`
`receipt_config.cut_type`	`FULL`
`receipt_config.delimiter_char`	`-`
`receipt_config.before_print_feed`	`3`
`receipt_config.before_cut_feed`	`5`
`receipt_config.encoding`	`CP1251`
`receipt_config.receipt_ext_format`	`true`
`receipt_config.receipt_print_vat`	`true`
`auto_z_report_enabled`	`false`
`allow_empty_z_report`	`false`
`auto_z_report_virtual`	`true`
`auto_backup_enabled`	`true`
`global_settings_update_interval_sec`	`1800`
`max_fm_blocks_warning_threshold`	`100`

Notes

- Swagger remains the contract source of truth for request/response shapes.
- This page is the behavioral reference for what the fields mean in practice.
- `eth_device` and `auto_backup_enabled` are visible in code/runtime behavior and are worth documenting here even where Swagger coverage is currently incomplete or newer than the older index page.
- `/config` and `/system/network-config` serve different purposes: `/config` controls service/runtime behavior, while `/system/network-config` controls Ethernet boot behavior.

On-device Diagnostic Menu

The diagnostic menu is available on ACLAS ARM devices only. It is accessed by holding the power button for approximately 3 seconds. The device prints the menu options on a non-fiscal receipt. The feed button cycles through the options, and the power button executes the selected option.

Menu options

#	Action	Description
1	X report	Prints the current X (day) report.
2	Diagnostic	Prints a diagnostic receipt with device info, firmware version, IP address, network mode, FM status, and similar.
3	Registration	Prints the current NRA registration data.
4	All fonts	Prints a font test sheet covering all available character sets.
5	FM report	Prints the fiscal memory (FM) block report.
6	Network: DHCP	Sets `NETWORK_MODE=dhcp` in the persistent network config. Takes effect on next reboot.
7	Network: static IP	Sets `NETWORK_MODE=default_static` in the persistent network config. Takes effect on next reboot.
8	Exit	Exits the menu and returns to normal operation.

Network mode options

Options 6 and 7 write directly to the persistent network config file (the same file managed by `PUT /system/network-config`). The change is saved immediately and the device reboots automatically (~2 seconds after the confirmation receipt is printed). The new mode is applied on boot when `app.sh` runs `setup_eth.sh`.

The current network mode is visible in the diagnostic receipt printed by option 2.

Implementation

The menu is implemented in `src/utils/diag_menu.c` and is compiled only when `HOST_CPU_ARM` is defined. It depends on the power button module (`src/utils/power_button/`) and the localization subsystem.

Related Docs

- API Overview
- Constraints and Behavior
- OpenAPI Reference Guide

Owner: Backend/Embedded team Last updated: 2026-04-21

Exports

Purpose

Describe how export endpoints are used by client integrations and what each KLEN export mode is for.

Scope

This page explains export behavior and integration guidance. Exact query parameters, response headers, content types, and schemas are defined in `doc/swagger.yml` and the runtime Swagger UI at `/resources/api`.

Audience

- Integrators downloading device data for backup, reconciliation, or audit workflows
- QA engineers validating export behavior
- Support tooling developers

Export Endpoints

The export surface contains:

- `/exports/fiscal-memory`
- `/exports/klen`

All export responses are intended to be consumed as files. Clients should honor the `Content-Disposition` response header when choosing the downloaded filename.

KLEN Export

`GET /exports/klen` exports journal/KLEN data without printing a journal report receipt and without intentionally changing KLEN, receipt, printer, or fiscal state.

The endpoint streams non-binary exports in internal chunks. Clients can download large matching ranges without the service building the whole result in memory first.

Formats

Use `resultFormat` for KLEN file format selection, matching the `/journal` result-format parameter:

- `resultFormat=json` or omitted Streams matching journal entries as JSON and returns `klen-export.json`.
- `resultFormat=csv` Streams KLEN-style CSV sale rows and returns `klen-export.csv`.
- `resultFormat=formatted_text` Streams printable journal text, aligned with `/journal?resultFormat=formatted_text`, and returns `klen-export.txt`.
- `resultFormat=binary` Returns the full SQLite database backup as `nsys-fiscal.db`.

`/exports/klen` does not read or validate the `format` query parameter.

Filters

JSON, CSV, and formatted-text KLEN exports support journal-style filters:

- `fromId` and `toId` for an inclusive journal ID range
- `fromDateTime` and `toDateTime` for an inclusive local timestamp range

- `type` for journal type filtering, for example `JOURNAL_FISCAL_RECEIPT`

ID range and date-time range are mutually exclusive. A request must not combine `fromId` / `toId` with `fromDateTime` / `toDateTime`.

If no range is supplied, the export streams all matching entries.

Binary Exception

`resultFormat=binary` intentionally ignores journal filters and always returns the complete database image. Use it for backup or support workflows, not for filtered journal readback.

CSV Scope

CSV export is item-oriented. It produces rows for sale-like journal records:

- fiscal receipts
- invoices
- storno receipts

Other journal entry types may be selected by filters, but they do not produce CSV sale rows.

Fiscal Memory Export

`GET /exports/fiscal-memory` exports fiscal memory either as structured JSON or as a binary fiscal-memory image.

Use:

- `format=json` or omitted for the structured `{ provision, records }` response
- `format=binary` for the raw fiscal-memory binary image

Related Docs

- openapi.md
- api-overview.md
- constraints-and-behavior.md

Owner: Backend/Embedded team Last updated: 2026-05-19

OpenAPI Reference Guide

Purpose

Point API consumers to the canonical contract artifact and explain how the Markdown documentation relates to it.

Scope

Navigation and source-of-truth guidance only.

Audience

- Developers who need exact endpoint definitions
- Reviewers checking whether a consumer-facing change is documented correctly

Canonical Contract

- Runtime Swagger/OpenAPI reference UI: /resources/api

Usage Rule

- Use /resources/api when you want the browser-rendered Swagger/OpenAPI reference from a running device.
- For exact paths, methods, query parameters, request bodies, and response schemas, use `doc/swagger.yml`.
- Use the Markdown files in `doc/public/` for quick orientation, behavior notes, and workflow guidance.
- Do not rely on historical Markdown endpoint lists as a canonical source.

How To Use The Specs In Practice

- Open /resources/api first if you want the interactive Swagger/OpenAPI view served by the device.
- For day-to-day integration work, start with `/resources/api`.
- The runtime API reference at `/resources/api` is the browser-facing Swagger UI for the local device OpenAPI contract.
- Use Swagger UI, Redoc, or any OpenAPI-capable tooling if you want a rendered view; the source of truth remains the YAML file in the repository and the runtime `/resources/api` entrypoint.
- Use the Markdown files in `doc/public/` first when you need lifecycle, constraints, and endpoint-purpose guidance, then switch to the OpenAPI file for exact request and response details.

Related Docs

- README.md
- api-overview.md
- constraints-and-behavior.md

Owner: Backend/Embedded team Last updated: 2026-03-31

Local Simulation For Integrators

Purpose

Explain the recommended local development environment for API integrators before they connect to a real device.

Recommendation

For most development work, start with the Docker-based simulation environment rather than a real device.

Practical rule:

- Use the simulator for the large majority of client development work.
- Switch to real hardware only for final device-specific validation, printer-dependent checks, or operational acceptance testing.

This should be the default path for almost all new integrations.

What The Simulator Provides

The Docker simulation image gives developers a close-to-real integration target in a single container:

- a runnable build of the fiscal REST API
- a mock NRA server
- persistent runtime state inside a Docker volume

This makes it useful for:

- early client development
- request/response validation
- registration-flow development
- NRA-related behavior checks
- regression testing before validation on real hardware

Why It Matters

For integrators, the simulator removes much of the friction associated with working with a physical device:

- no immediate dependency on target hardware
- repeatable local startup
- isolated state
- easier debugging and retrying of flows
- NRA mock behavior available in the same environment

Where The Detailed Usage Lives

The simulator is built using repository tooling, but the integration documentation should remain usable even when the deployed device documentation does not include the `tools/` directory.

In practice, the simulator workflow is:

1. build an installable x64 package of the service
2. build the Docker simulation image from the repository

3. run the container with ports for the fiscal REST API and the NRA mock server exposed locally

The simulator uses one container that starts both the fiscal service and the NRA mock server together.

What To Expect From The Image

The image runs both:

- the local fiscal service
- the NRA mock server

Typical local ports:

- `8888` for the fiscal REST API
- `7000` for the NRA mock server

Typical local smoke test:

- call `/hello` on the fiscal service
- call the NRA mock status endpoint on port `7000`

When To Use Real Hardware Instead

Move from the simulator to a real device when you need to verify:

- physical printer behavior
- device-specific provisioning/setup details
- hardware/network/modem specifics
- final acceptance behavior on the target platform

Related Docs

- README.md
- integration-flows.md
- openapi.md

Owner: Backend/Embedded team Last updated: 2026-04-01