



[Onboarding di base 2 di 2]

Sistema
Domotico su
piattaforma
Arduino



AUTORE: ANDREA LANDO

REVISIONE 1.0.0

DATA ULTIMA REVISIONE: 22.05.2026

Sommario

Prefazione	9
Obiettivo di questo documento.....	9
Il ruolo del Frontend	9
Architettura Avanzata del Frontend	10
Obiettivo del capitolo	10
Filosofia del Frontend	10
Ruolo del Frontend	11
Flusso di avvio del Frontend	12
Struttura interna del Frontend	13
Perché il Frontend è fondamentale	13
Diagramma generale del Frontend.....	14
Conclusione del capitolo.....	14
WatchEngine (Diagnostics Aree)	16
Obiettivo del capitolo	16
Ruolo del WatchEngine	16
Come funziona.....	16
Configurazione nel FrontendConfig.....	16
Cosa monitora il WatchEngine	17
Log diagnostici generati.....	18
Quando usare il WatchEngine	18
Best practice	18
Diagramma del flusso WatchEngine.....	19
Conclusione del capitolo.....	19
HVACEngine	20
Obiettivo del capitolo	20
Architettura del modulo HVAC	20
Configurazione HVAC nel FrontendConfig.....	21
Flusso operativo del Task_HVAC	22
Zone HVAC.....	23
Modalità operative della pompa di calore	23
Logica finestra aperta	24
Logica sicurezza temperatura esterna.....	24
Logica ventilazione	24
Logica fan-coil per zona	24
Logica pompa di circolazione.....	24
Logica ACS (acqua calda sanitaria).....	25

Logica defrost	25
Diagnostica HVAC	26
Diagramma del flusso HVAC	26
Tipologie di impianto supportate dal modulo HVAC	27
Impianto con 1 accumulo acqua tecnica + 1 accumulo sanitario (DHW) con valvola a 3 vie	27
Impianto con solo accumulo tecnico (senza ACS)	28
Impianto con fan-coil diretti (senza puffer)	28
Impianto radiante (pavimento/soffitto) con miscelatrice	29
Impianto ibrido (pompa di calore + caldaia)	29
Tabella riassuntiva compatibilità	30
Conclusione	30
WeatherEngine	32
Obiettivo del capitolo	32
Architettura del modulo meteo	32
6.2. Configurazione nel FrontendConfig	33
WeatherEngine (frontend)	34
WeatherStation — motore meteo puro	35
Eventi meteo generati	36
Allarmi meteo generati	36
Integrazione con automazioni	37
Diagramma del flusso meteo	37
Esempio di automazione reale	38
Conclusione	38
PowerSupervisorEngine	39
Obiettivo del modulo	39
Architettura del modulo	39
Configurazione nel FrontendConfig	40
Setup del modulo	41
Lettura dei segnali dal buffer	41
Logica di supervisione (motore puro)	41
Extra signals (estensione potente)	42
Loop non bloccante	42
Integrazione con automazioni	42
Diagnostica	43
Diagramma del flusso	43
Quando usare PowerSupervisorEngine	43
Conclusione del capitolo	44

PowerEngine	45
Obiettivo del modulo	45
Architettura del modulo	45
collega callback	46
8.3. Configurazione nel FrontendConfig	46
Setup del modulo	47
Flusso operativo del PowerEngine	47
PVStringManager — gestione fotovoltaico	48
BatteryManager — gestione batteria	48
LoadManager — gestione carichi	49
LimitManager — protezione limiti	49
AutoTuneManager — auto-ottimizzazione	50
Esempio reale di ciclo Power	50
Integrazione con automazioni	50
Diagnostica	51
Diagramma del flusso PowerEngine	51
Conclusione del capitolo	51
SecuritySensorEngine	53
Obiettivo del modulo	53
Architettura del modulo	53
Configurazione nel FrontendConfig	54
Il cuore del sistema: WiredSensorsManager	54
SensorChannel — il singolo canale	55
Sensor — aggregazione canali	55
ZoneManager — gestione zone	56
AlarmDispatcher — callback dinamiche	56
SecurityOrchestrator — orchestratore backend	56
SystemManager — bitmask sicurezza	57
SecuritySensorEngine (frontend)	57
Loop del SecuritySensorEngine	57
Integrazione con automazioni	58
Diagnostica	58
Diagramma del flusso	59
Conclusione	59
JobsEngine	60
Obiettivo del modulo	60
Architettura del modulo	60

AsyncScheduler — il cuore del sistema	61
Tipologie di Step	62
Priorità dei job	62
JobsEngine — implementazione frontend	62
10.7. Condizioni e skip	64
Costruzione dei job	64
Esecuzione del JobsEngine	64
Integrazione con automazioni	64
Esempio di job avanzato (consigliato)	65
Diagnostica	65
Conclusione del capitolo.....	66
TaskEngine	67
Obiettivo del modulo.....	67
Architettura del TaskEngine	67
Perché il TaskEngine è fondamentale.....	67
Registrazione dei task.....	68
Esecuzione del ciclo	68
FrontendCycleDone	69
Tipologie di task.....	69
Esempio reale di task.....	69
Determinismo del ciclo	70
Integrazione con JobsEngine	70
Integrazione con automazioni	70
Diagramma del flusso TaskEngine	71
Conclusione	71
AutomationEngine (AUTOMATION_JSON)	72
Obiettivo del modulo.....	72
Architettura generale	72
Scene — blocchi di azioni	73
Rule — regole classiche	73
ScheduledRule — regole temporali.....	74
DynamicAutomation — logiche avanzate	74
ActionSequence — sequenze temporizzate	75
CompositeRule — logiche complesse multi-input.....	76
AutomationBuilder — pipeline di costruzione	76
Ciclo di esecuzione dell'AutomationEngine.....	77
Integrazione con gli altri moduli	77

Diagnostica	78
Conclusione del capitolo.....	78
Best Practices per AUTOMATION_JSON	79
Principio fondamentale: tutto passa dal Buffer	79
Usa SEMPRE scene, mai azioni dirette	79
Usa “NoAction” quando non vuoi fare nulla	79
Non duplicare logiche già presenti nei moduli	79
Usa CompositeRule per logiche complesse	80
Usa TrendRule solo quando serve davvero	80
15.7. Usa debounce e TOF per evitare oscillazioni.....	81
Usa MultiRule solo per condizioni semplici	81
Usa ScheduledRule per orari, mai delay manuali	81
Le sequenze devono essere brevi e deterministiche.....	81
Evita scene con troppi comandi.....	82
Usa nomi scena coerenti e leggibili	82
Evita condizioni su aree non inizializzate.....	82
Evita regole con intervalli troppo brevi	83
Usa “NoAction” per evitare log inutili	83
Non duplicare scene o regole	83
Usa ActionSequence per animazioni, non per logica	83
Mantieni il JSON leggibile	83
Test incrementale	83
Conclusione del capitolo.....	84
Esempi completi di AUTOMATION_JSON	85
Struttura generale del file AUTOMATION_JSON	85
Esempio 1 — Automazioni HVAC (Cappa Cucina + Bagno Umido).....	86
Scene.....	86
Regole dinamiche (CompositeRule).....	87
Esempio 2 — Automazioni Security (Intrusione + Finestre Aperte)	89
Scene.....	89
Regola bitmask (intrusione).....	89
MultiRule (finestre aperte + porta aperta)	90
Esempio 3 — Automazioni Power (Load Shedding)	91
Scene.....	91
13.5. Esempio 4 — Sequenza temporizzata (Luci scenografiche)	92
Conclusione	92
Disclaimer Legale – Limitazione di Responsabilità	94

Prefazione

Obiettivo di questo documento

Il ruolo del Frontend

Nel primo onboarding hai visto che:

- **Main** costruisce FrontendConfig e chiama il Frontend.
- **DMSetup.hpp** è la “mappa dell’impianto” (dispositivi, buffer, watchdog, HVAC, Power, Weather, Security, MQTT, ecc.).
- **DomoManager** è il cuore operativo (Modbus, buffer, automazioni, watchdog, medie, ecc.).

Il **Frontend** è il “direttore d’orchestra” che:

- prende FrontendConfig
- inizializza i motori di dominio (HVAC, Weather, Power, Security, PowerSupervisor, medie, Jobs, comunicazioni...)
- registra i task nel TaskEngineOrchestrator
- esegue il ciclo deterministico non bloccante.

In pratica: **tutta la logica di alto livello è nei motori di dominio**, ma **tutta la configurazione è in FrontendConfig / DMSetup.hpp / AUTOMATION_JSON**.

Architettura Avanzata del Frontend

Obiettivo del capitolo

Questo capitolo introduce in profondità l'architettura del **Frontend Engine**, il livello che orchestra tutti i moduli funzionali del sistema DomoManager. Se nel documento 1 di 2 hai visto la visione generale, qui entriamo nel dettaglio operativo:

- come il Frontend costruisce il sistema
- come inizializza i motori di dominio
- come registra i task
- come gestisce il ciclo deterministico
- come interagisce con DomoManager
- come si integra con la configurazione dichiarativa (FrontendConfig + DMSetup.hpp)

Il Frontend è il cuore dell'architettura moderna di DomoManager: **non contiene logica applicativa**, ma **coordina** i moduli che la implementano.

Filosofia del Frontend

Il Frontend è progettato secondo tre principi fondamentali:

1. Tutto è configurazione, niente è hardcoded

Il Frontend non contiene valori fissi, non interpreta sensori, non decide soglie, non gestisce automazioni. Tutto proviene da:

- FrontendConfig (costruito nel main)
- DMSetup.hpp (mappa dell'impianto)
- AUTOMATION_JSON (logiche dinamiche)

2. Ogni modulo è indipendente

Ogni Engine (HVAC, Weather, Power, Security, ecc.) è:

- autonomo
- configurabile
- sostituibile
- isolato dagli altri moduli

Il Frontend li inizializza e li fa girare, ma non ne conosce la logica interna.

3. Il ciclo principale non blocca mai

Il Frontend usa un **TaskEngine** con:

- task rate-limited
- nessuna attesa bloccante
- round-robin deterministico

- priorità implicite basate sugli intervalli

Questo garantisce:

- reattività costante
- nessun freeze
- nessun jitter critico
- comportamento deterministico

Ruolo del Frontend

Il Frontend è il **livello di orchestrazione** tra:

- configurazione (FrontendConfig + DMSetup)
- motori di dominio (HVAC, Weather, Power, Security...)
- DomoManager (buffer, Modbus, automazioni, watchdog)
- comunicazioni (MQTT, Bridge, WebAPI)

Il suo compito è:

1. Inizializzare il sistema

- rete
- Modbus TCP
- dispositivi
- buffer
- diagnostica
- motori funzionali
- comunicazioni

2. Registrare i task

Ogni modulo viene eseguito con un intervallo configurabile:

- HVAC → ogni X ms
- Meteo → ogni Y ms
- Power → ogni Z ms
- Security → ogni N ms
- Averages → ogni M ms
- Comunicazioni → round-robin interno
- Jobs → ogni K ms

3. Eseguire il ciclo deterministico

Il Frontend chiama:

```
TaskEngineOrchestrator::Loop()
```

che esegue solo i task scaduti, senza bloccare.

4. Fornire callback di alto livello

Il Frontend riceve eventi dai motori:

- pioggia → chiudi finestre
- vento forte → chiudi finestre
- allarme intrusion → scrivi bitmask
- superamento limite potenza → suggerimenti
- fault alimentazione → log + eventuali scene
- variazioni AEE → invio al Bridge

e li trasforma in:

- log diagnostici
- scene di automazione
- aggiornamenti buffer
- notifiche MQTT
- comandi verso dispositivi

Flusso di avvio del Frontend

Il flusso reale è:

1. Il main costruisce la configurazione

```
FrontendConfig cfg;
```

2. Il main avvia il Frontend

```
DomoManagerFrontendEngine::Setup(cfg);
```

3. Il Frontend:

- inizializza rete
- costruisce DomoManager
- applica DMSetup (aree, dispositivi, toggles, splits, routes)
- inizializza i motori:
 - WatchEngine
 - HVACEngine
 - WeatherEngine
 - PowerEngine
 - PowerSupervisorEngine

- SecuritySensorEngine
- Averages
- Jobs
- WebAPI / MQTT / Bridge
- registra i task nel TaskEngine

4. Il main entra nel loop

```
DomoManagerFrontendEngine::Loop();
```

5. Il ciclo deterministico parte

Ogni task viene eseguito al proprio intervallo.

Struttura interna del Frontend

Il Frontend è composto da:

Motori funzionali

- **WatchEngine** → diagnostica aree
- **HVACEngine** → climatizzazione multi-zona
- **WeatherEngine** → eventi meteo + automazioni finestre
- **PowerEngine** → gestione carichi, limiti, forecast
- **PowerSupervisorEngine** → supervisione alimentazione
- **SecuritySensorEngine** → sensori cablati + bitmask
- **JobsEngine** → scheduler asincrono
- **AveragesEngine** → medie dinamiche

Motori di comunicazione

- **MQTTEngine**
- **WebAPIEngine**
- **BridgeEngine** (via DMAAdapters)

Motore di orchestrazione

- **TaskEngine**
- **TaskEngineOrchestrator**

Collante finale

- **DomoManagerFrontendEngine**

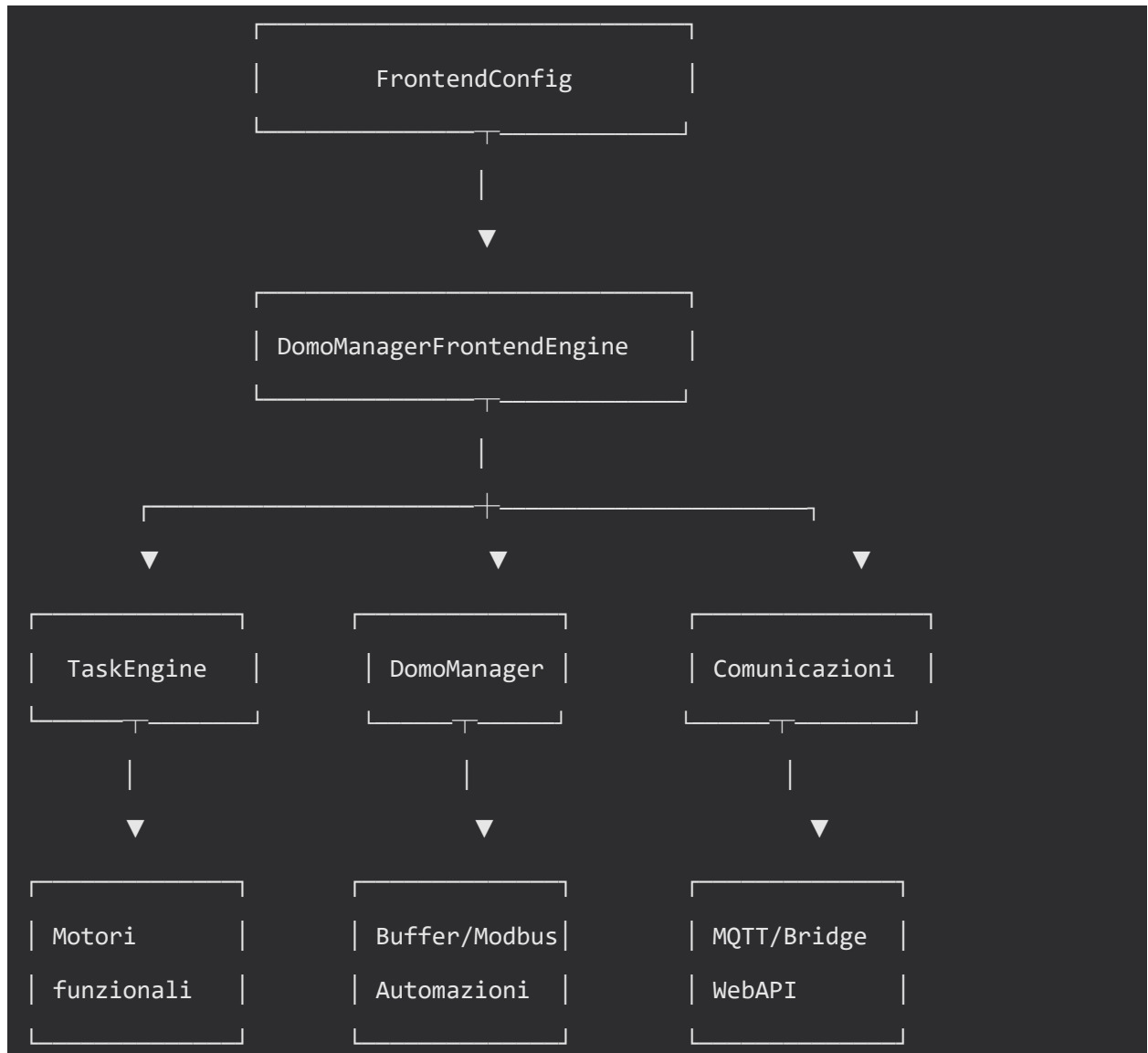
Perché il Frontend è fondamentale

Il Frontend:

- separa completamente **logica** e **configurazione**

- permette aggiornamenti firmware senza toccare la logica
- permette di aggiungere moduli senza modificare il core
- garantisce un ciclo deterministico
- centralizza la diagnostica
- fornisce un punto unico di orchestrazione
- mantiene il sistema modulare, scalabile e leggibile

Diagramma generale del Frontend



Conclusione del capitolo

Il Frontend è il livello che rende DomoManager:

- modulare
- scalabile
- dichiarativo

- non bloccante
- industriale
- estendibile

Nei capitoli successivi analizzeremo **ogni Engine** in dettaglio, mostrando:

- configurazione
- flusso dati
- callback
- interazione con DomoManager
- esempi reali
- best practice

WatchEngine (Diagnostica Aree)

Obiettivo del capitolo

Il WatchEngine è il modulo più semplice del Frontend, ma anche uno dei più importanti in fase di commissioning e diagnostica. Serve a monitorare un insieme di aree del buffer e a rilevare:

- aree che non cambiano mai
- aree che cambiano troppo spesso
- aree che assumono valori anomali
- aree che non sono coerenti con la configurazione dell'impianto

Il WatchEngine non modifica il comportamento del sistema: **fornisce solo diagnostica avanzata**, utile per installatori e sviluppatori.

Ruolo del WatchEngine

Il WatchEngine:

- legge la configurazione FrontendConfig::Watch
- registra nel DomoManager le aree da monitorare
- abilita la diagnostica interna del modulo Watch
- non ha un loop dedicato (la diagnostica è gestita da DomoManager)
- non genera eventi, non modifica valori, non esegue logiche

È un modulo **read-only**, completamente passivo.

Come funziona

Il WatchEngine si limita a chiamare:

```
auto& diag = manager.getWatchDiag();  
diag.addArea(cfg.aree[i]);
```

Per ogni area registrata, il modulo Watch del DomoManager:

- tiene traccia degli ultimi valori
- calcola variazioni, spike, anomalie
- registra tempi di aggiornamento
- rileva aree mai inizializzate
- rileva aree scritte più volte in modo incoerente
- produce log diagnostici dettagliati

Il WatchEngine è quindi un **abilitatore** della diagnostica interna del DomoManager.

Configurazione nel FrontendConfig

La sezione dedicata è:

```
struct Watch {
    bool enabled = false;
    const int* aree = nullptr;
    size_t count = 0;
} watch;
```

Parametri

Campo	Descrizione
enabled	abilita/disabilita il modulo
aree	array di ID di aree del buffer
count	numero di aree da monitorare

Esempio reale

Nel main:

```
static const int watchAreas[] = { AREA_DI_BAGNO, AREA_D01, AREA_D02 };

cfg.watch.enabled = true;
cfg.watch.aree = watchAreas;
cfg.watch.count = 3;
```

Cosa monitora il WatchEngine

Il modulo Watch del DomoManager (non il WatchEngine) monitora:

1. Frequenza di aggiornamento

- area che non cambia mai → possibile errore Modbus
- area che cambia troppo spesso → rimbalzi, disturbi, cablaggio errato

2. Spike e valori anomali

- valori fuori range
- valori incoerenti con la soglia configurata
- valori negativi dove non previsti

3. Aree mai inizializzate

- area definita ma mai scritta
- area scritta più volte da moduli diversi

4. Coerenza con la configurazione

- aree forward non aggiornate
- aree virtuali non coerenti

- aree duplicate

Log diagnostici generati

Quando un'area monitorata presenta anomalie, il DomoManager produce log come:

```
[WATCH] Area 14 changed too frequently  
[WATCH] Area 10 never updated  
[WATCH] Area 23 spike detected: 0 → 255  
[WATCH] Area 17 inconsistent write source
```

Questi log sono fondamentali per:

- collaudo impianto
- debugging Modbus
- verifica cablaggio
- validazione configurazione DMSetup.hpp
- analisi prestazioni

Quando usare il WatchEngine

Consigliato in:

- impianti nuovi
- fasi di commissioning
- debugging Modbus
- test di automazioni
- validazione configurazione buffer

Non necessario in:

- impianti stabili e già collaudati
- sistemi con molte aree (per evitare log eccessivi)

Best practice

1. Monitorare solo le aree critiche

Esempi:

- ingressi digitali (pulsanti, sensori)
- uscite digitali (relè, attuatori)
- aree di stato (bitmask security, power state)

2. Evitare di monitorare aree ad alta frequenza

Esempi:

- sensori analogici che cambiano ogni 100 ms

- medie dinamiche
- trend

3. Usare il WatchEngine per validare DMSetup.hpp

Se un'area non cambia mai → probabilmente è mappata male.

Diagramma del flusso WatchEngine



Conclusione del capitolo

Il WatchEngine è un modulo semplice ma potentissimo per:

- validare l'impianto
- diagnosticare problemi
- verificare la configurazione
- individuare errori Modbus
- controllare la coerenza del buffer

È uno strumento essenziale per installatori e sviluppatori, soprattutto nelle prime fasi di messa in servizio.

HVACEngine

Obiettivo del capitolo

Questo capitolo descrive in modo completo il funzionamento del modulo HVAC di DomoManager, composto da:

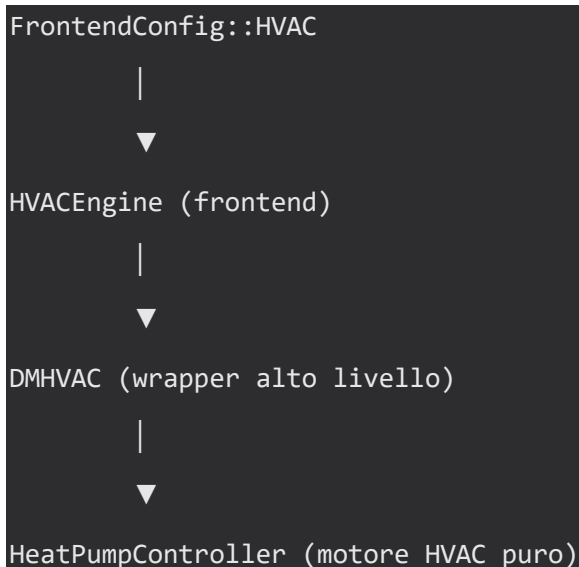
- **HVACEngine** (frontend)
- **DMHVAC** (wrapper alto livello)
- **HeatPumpController** (motore HVAC puro)

Il modulo HVAC è uno dei più complessi dell'intero sistema: gestisce la pompa di calore, le zone, i fan-coil, la logica ACS, la logica anti-legionella, la logica defrost, la sicurezza, la finestra aperta, la ventilazione e la pompa di circolazione.

Il tutto in modo **completamente dichiarativo**, senza logica hardcoded nel firmware.

Architettura del modulo HVAC

Il modulo HVAC è composto da tre livelli:



HVACEngine

- inizializza DMHVAC
- fornisce accesso al motore
- viene chiamato dal TaskEngine

DMHVAC

- costruisce HeatPumpController
- crea le zone
- applica la configurazione ACS
- applica la configurazione anti-legionella
- gestisce la valvola a 3 vie

- esegue il loop principale

HeatPumpController

- implementa tutta la logica HVAC:
 - riscaldamento
 - raffrescamento
 - AUTO multi-zona
 - defrost
 - sicurezza temperatura esterna
 - finestra aperta
 - ACS
 - anti-legionella
 - pompa di circolazione
 - ventilazione
 - fan-coil per zona

Configurazione HVAC nel FrontendConfig

La sezione HVAC del FrontendConfig eredita direttamente da DMHVAC::HVACConfig.

Esempio:

```
cfg.hvac.enabled = true;
cfg.hvac.intervalMs = 5000;

static DMHVAC::HVACConfig::Zone zones[] = {
    { "Zona Giorno", 21.0f, AREA_TEMP_GIORNO, 1 },
    { "Zona Notte", 20.0f, AREA_TEMP_NOTTE, 2 }
};

cfg.hvac.zones = zones;
cfg.hvac.zoneCount = 2;

cfg.hvac.readIndoorTemp = [](){ return buffer.getValueFast(100) / 10.0f; };
cfg.hvac.readOutdoorTemp = [](){ return buffer.getValueFast(101) / 10.0f; };
cfg.hvac.readWindowOpen = [](){ return buffer.getValueFast(102) != 0; };
```

```
cfg.hvac.initialMode = HeatPumpController::Mode::AUTO;  
cfg.hvac.initialSetpoint = 21.0f;
```

Parametri principali

Campo	Descrizione
zones[]	elenco zone HVAC
readIndoorTemp	lettura temperatura interna
readOutdoorTemp	lettura temperatura esterna
readWindowOpen	finestra aperta
initialMode	OFF / MANUAL / HEATING / COOLING / AUTO
hpConfig	parametri avanzati del motore

Flusso operativo del Task_HVAC

Il TaskEngine chiama:

```
HVACEngine::Loop(  
    now,  
    zoneTemps,  
    tInterna,  
    tEsterna,  
    finestraAperta,  
    ht  
);
```

Dove:

- zoneTemps[] → temperature per zona lette dal buffer
- tInterna → temperatura interna globale
- tEsterna → temperatura esterna
- finestraAperta → booleano
- ht → orario (giorno, ora, minuto)

Il DMHVAC:

1. aggiorna le temperature delle zone
2. aggiorna indoor/outdoor
3. aggiorna finestra aperta
4. esegue logica ACS
5. esegue logica anti-legionella

6. esegue il controlLoop del motore HVAC

Zone HVAC

Ogni zona è un oggetto ZoneHVAC:

- nome
- setpoint
- temperatura
- fan-coil associato
- richiesta riscaldamento
- richiesta raffrescamento

La logica di richiesta è:

```
heatRequest = (temperature < setpoint - hysteresis);  
coolRequest = (temperature > setpoint + hysteresis);
```

Modalità operative della pompa di calore

OFF

- compressore OFF
- ventilazione LOW
- pompa di circolazione OFF

MANUAL

- compressore ON
- ventilazione HIGH
- pompa ON

HEATING

- logica riscaldamento con isteresi

COOLING

- logica raffrescamento con isteresi

AUTO

- valuta tutte le zone:
 - se almeno una zona richiede calore → HEATING
 - se almeno una zona richiede freddo → COOLING

DEFROST

- attivato automaticamente se T esterna < soglia
- durata configurabile

- compressore OFF
- ventilazione MEDIUM
- pompa ON

Logica finestra aperta

Se windowOpen == true:

- parte un timer
- se supera windowOpenTimeoutMs:
 - compressore OFF
 - ventilazione LOW
 - pompa ON

Logica sicurezza temperatura esterna

Se T esterna < minOutdoorTemp o > maxOutdoorTemp:

- safetyActive = true
- compressore OFF
- pompa ON

Quando la temperatura rientra nel range → safety disattivata.

Logica ventilazione

La velocità della ventola dipende da:

- differenza tra setpoint e temperatura
- isteresi della ventola

```
diff < diffLow → LOW
diff < diffMed → MEDIUM
diff > diffMed → HIGH
```

Logica fan-coil per zona

Ogni zona ha un fan-coil associato.

Il motore attiva il fan-coil solo se:

- la zona richiede calore e la pompa è in HEATING
- la zona richiede freddo e la pompa è in COOLING

Il callback è:

```
fanCoilCallback(nomeZona, fanCoilId, stato);
```

Logica pompa di circolazione

La pompa si attiva quando:

- il compressore è ON
- oppure entro postCirculationMs dallo spegnimento

E si disattiva solo dopo:

- minCirculationCycleMs

Logica ACS (acqua calda sanitaria)

Il motore gestisce:

- priorità ACS
- setpoint ACS
- valvola a 3 vie
- anti-legionella
- temperatura ACS

Normal DHW

Se $T_{ACS} < \text{setpoint} - 2^{\circ}\text{C}$:

- target = DHW
- modalità = HEATING

Anti-legionella

Attivata:

- in un giorno specifico
- a un'ora specifica
- per una durata massima
- fino al raggiungimento del setpoint anti-legionella

Logica defrost

Se $T_{\text{esterna}} < \text{defrostThreshold}$:

- entra in DEFROST
- compressore OFF
- ventilazione MEDIUM
- pompa ON
- durata = defrostDurationMs

Diagnostica HVAC

Il motore fornisce una diagnostica completa:

```
===== HVAC DIAGNOSTIC =====  
  
Mode: AUTO  
  
Indoor Temp: 21.3 °C  
Outdoor Temp: 7.2 °C  
Setpoint: 21.0 °C  
Compressor: OFF  
Fan: LOW  
Hydraulic Target: SPACE_HEATING  
  
Zones:  
- Zona Giorno | T=21.3°C | SP=21.0°C | NONE | FanCoil=OFF  
- Zona Notte | T=19.8°C | SP=20.0°C | HEAT | FanCoil=ON  
  
===== END HVAC DIAGNOSTIC =====
```

Diagramma del flusso HVAC

```
Task_HVAC  
  |  
  ▼  
HVACEngine::Loop()  
  |  
  ▼  
DMHVAC::loop()  
  |  
  ▼  
HeatPumpController  
  ├── sicurezza T esterna  
  ├── finestra aperta  
  ├── defrost  
  ├── ACS  
  ├── anti-legionella  
  └── AUTO multi-zona
```

```
└─ ventilazione
└─ fan-coil
└─ pompa di circolazione
```

Tipologie di impianto supportate dal modulo HVAC

Il motore HVAC di DomoManager (HeatPumpController + DMHVAC) è stato progettato per essere **agnostico rispetto all'impianto fisico**, purché la logica possa essere espressa tramite:

- setpoint
- temperature di zona
- temperatura interna/esterna
- stato finestra
- ACS (acqua calda sanitaria)
- valvola a 3 vie
- fan-coil
- pompa di circolazione

Questo lo rende compatibile con **tutte le principali configurazioni impiantistiche residenziali e light-commercial**.

Di seguito trovi i **4 scenari più comuni**, con **schemi funzionali** e note operative.

Impianto con 1 accumulo acqua tecnica + 1 accumulo sanitario (DHW) con valvola a 3 vie

Questo è lo scenario più diffuso nelle abitazioni moderne con pompa di calore.

Componenti tipici

- Pompa di calore monoblocco o split
- Accumulo acqua tecnica (puffer)
- Accumulo sanitario (bollitore)
- Valvola a 3 vie deviatrice
- Pompa di circolazione
- Fan-coil o pannelli radianti
- Sensore temperatura ACS
- Sensore temperatura puffer

Come lavora il modulo HVAC

- La valvola a 3 vie viene comandata tramite callback:
 - **DHW** → riscaldamento bollitore
 - **SPACE_HEATING** → riscaldamento/raffrescamento ambienti
- La logica ACS gestisce:
 - priorità sanitaria

- setpoint ACS
- anti-legionella
- durata massima ciclo
- La pompa di circolazione segue la logica del compressore
- Le zone HVAC comandano i fan-coil

Quando usare questa configurazione

- Abitazioni con pompa di calore aria-acqua
- Impianti misti fan-coil + radiante
- Impianti con bollitore sanitario dedicato

Impianto con solo accumulo tecnico (senza ACS)

Componenti

- Pompa di calore
- Puffer tecnico
- Pompa di circolazione
- Fan-coil o radiante

Come lavora il modulo HVAC

- La sezione ACS viene disattivata (`cfg.acs.enabled = false`)
- La valvola a 3 vie non è necessaria
- Il motore gestisce:
 - riscaldamento
 - raffrescamento
 - defrost
 - sicurezza T esterna
 - finestra aperta
 - ventilazione
 - zone

Quando usarlo

- Impianti senza bollitore sanitario
- Impianti con ACS prodotta da boiler elettrico o gas separato

Impianto con fan-coil diretti (senza puffer)

Componenti

- Pompa di calore
- Fan-coil diretti
- Pompa di circolazione
- Sensori di temperatura per zona

Come lavora il modulo HVAC

- Le zone HVAC comandano direttamente i fan-coil
- Il compressore segue la logica multi-zona
- La pompa di circolazione è gestita automaticamente
- Nessuna valvola a 3 vie
- ACS opzionale

Quando usarlo

- Appartamenti con solo fan-coil
- Impianti senza inerzia termica
- Ristrutturazioni leggere

Impianto radiante (pavimento/soffitto) con miscelatrice

Componenti

- Pompa di calore
- Collettore radiante
- Valvola miscelatrice (non gestita direttamente dal modulo HVAC)
- Pompa di circolazione
- Sensori di temperatura ambiente

Come lavora il modulo HVAC

- Il modulo HVAC gestisce:
 - setpoint ambiente
 - logica multi-zona
 - finestra aperta
 - sicurezza
 - ACS
- La miscelatrice è gestita da:
 - centralina esterna
 - o automazione personalizzata via buffer

Quando usarlo

- Abitazioni con impianto radiante
- Impianti con inerzia elevata
- Necessità di comfort termico stabile

Impianto ibrido (pompa di calore + caldaia)

Componenti

- Pompa di calore
- Caldaia a gas

- Valvola di commutazione
- Accumulo tecnico
- ACS opzionale

Come lavora il modulo HVAC

Il modulo HVAC può gestire:

- la parte pompa di calore
- la logica ACS
- la logica zone

La commutazione caldaia/PdC può essere gestita tramite:

- automazioni JSON
- logiche su buffer
- regole dinamiche

Quando usarlo

- Climi freddi
- Impianti esistenti con caldaia
- Ristrutturazioni parziali

Tabella riassuntiva compatibilità

Tipo impianto	HVAC	ACS	Anti-legionella	3 vie	Fan-coil	Radiante	Note
Puffer + Bollitore	✓	✓	✓	✓	✓	✓	Configurazione ideale
Solo Puffer	✓	✗	✗	✗	✓	✓	Semplice e stabile
Fan-coil diretti	✓	opz	opz	✗	✓	✗	Reattivo
Radiante	✓	opz	opz	✗	✗	✓	Inerzia elevata
Ibrido PdC + Caldaia	✓	opz	opz	opz	✓	✓	Richiede automazioni

Conclusione

Il modulo HVAC di DomoManager è progettato per funzionare con **qualsiasi impianto idronico moderno**, grazie alla sua architettura:

- modulare
- dichiarativa
- multi-zona
- estensibile
- indipendente dall'hardware

Gli schemi mostrati rappresentano i casi più comuni, ma il motore può essere adattato a configurazioni personalizzate tramite:

- FrontendConfig

- automazioni JSON
- callback personalizzate
- logiche su buffer

Il Frontend non contiene logica HVAC: **tutto è delegato al motore HeatPumpController**, configurato tramite FrontendConfig.

WeatherEngine

Obiettivo del capitolo

Il modulo meteo di DomoManager è composto da due livelli:

- **WeatherEngine** → frontend
- **WeatherOrchestrator + WeatherStation** → backend (motore meteo puro)

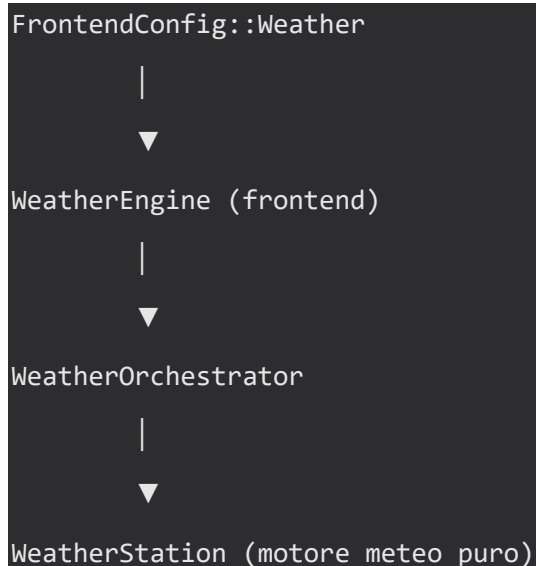
Questo capitolo descrive:

- come il Frontend inizializza il modulo meteo
- come il motore WeatherStation elabora i dati
- come vengono generati eventi (RainStart, WindGustStart, DayStart...)
- come vengono generati allarmi (TempLow, WindHigh...)
- come il frontend può reagire agli eventi (es. chiusura finestre)
- come configurare correttamente il modulo meteo
- come integrarlo con automazioni e scenari reali

Il modulo meteo è uno dei più completi dell'intero sistema: implementa **sensor fusion**, **filtri O(1)**, **debounce**, **isteresi**, **eventi**, **allarmi**, **diagnostica**.

Architettura del modulo meteo

Il flusso è:



WeatherEngine

- registra callback eventi
- registra callback allarmi
- inizializza WeatherStation
- chiama `WeatherOrchestrator::Loop()`

WeatherStation

- legge sensori
- applica filtri
- rileva pioggia
- rileva raffiche
- rileva giorno/notte
- genera eventi
- genera allarmi
- fornisce diagnostica

6.2. Configurazione nel FrontendConfig

La sezione meteo è:

```
struct Weather {  
    bool enabled = false;  
    uint32_t intervalMs = 5000;  
    WeatherStation::Config config;  
    bool autoCloseWindows;  
} weather;
```

Parametri principali

Campo	Descrizione
enabled	abilita il modulo meteo
intervalMs	frequenza aggiornamento
config	configurazione completa WeatherStation
autoCloseWindows	se true, il frontend chiude finestre in caso di pioggia/vento

Esempio reale

```
cfg.weather.enabled = true;  
cfg.weather.intervalMs = 3000;  
  
cfg.weather.config.common.readTemp = [](){ return analogRead(A0); };  
cfg.weather.config.common.readWind = [](){ return analogRead(A1); };  
cfg.weather.config.common.readRain = [](){ return analogRead(A2); };  
cfg.weather.config.common.readLight = [](){ return analogRead(A3); };
```

```

cfg.weather.config.common.tempFactor = 0.1f;
cfg.weather.config.common.windFactor = 0.5f;
cfg.weather.config.common.rainFactor = 1.0f;
cfg.weather.config.common.lightFactor = 1.0f;

cfg.weather.config.common.lowTempThreshold = 5.0f;
cfg.weather.config.common.highTempThreshold = 40.0f;
cfg.weather.config.common.highWindThreshold = 200.0f;
cfg.weather.config.common.highRainThreshold = 300.0f;

cfg.weather.config.lightDayThreshold = 200.0f;
cfg.weather.config.lightNightThreshold = 50.0f;

cfg.weather.autoCloseWindows = true;

```

WeatherEngine (frontend)

Il frontend è molto semplice:

```

static void Setup(const FrontendConfig::Weather& cfg) {
    WeatherOrchestrator::Setup(cfg, weatherEventHandler, weatherAlarmHandler);
}

```

Eventi gestiti dal frontend

- RainStart → chiudi finestre
- RainStop
- WindGustStart → chiudi finestre
- WindGustEnd
- DayStart
- NightStart → chiudi finestre

Allarmi gestiti dal frontend

- TempLow
- TempHigh
- WindHigh → chiudi finestre
- RainHigh → chiudi finestre

Il frontend **non interpreta i sensori**: riceve solo eventi e allarmi.

WeatherStation — motore meteo puro

Il motore meteo è estremamente avanzato. Implementa:

1. Sensor fusion

Legge 4 sensori:

- temperatura
- vento
- pioggia
- luce

Ogni sensore ha un fattore di scala configurabile.

2. Filtri O(1)

Usa finestre mobili di 10 campioni con somma incrementale:

- nessuna allocazione
- nessun ciclo di somma
- prestazioni costanti

3. Giorno / Notte con isteresi

Basato sulla luce:

Codice

`light >= dayThreshold` → DAY

`light <= nightThreshold` → NIGHT

4. Pioggia (RainStart / RainStop)

Con doppio debounce:

- soglia inizio pioggia
- soglia fine pioggia
- contatori indipendenti

5. Raffiche di vento (WindGustStart / WindGustEnd)

Basato su:

Codice

`delta = wind - lastWindValue`

Se delta supera la soglia → raffica.

6. Allarmi meteo

Con debounce:

- TempLow

- TempHigh
- WindHigh
- RainHigh

7. Diagnostica completa

Stampa:

- valori filtrati
- stati logici
- soglie
- allarmi

Eventi meteo generati

RainStart

Pioggia rilevata oltre soglia per N campioni.

RainStop

Pioggia sotto soglia per N campioni.

WindGustStart

Delta vento > soglia per N campioni.

WindGustEnd

Delta vento <= 0 per N campioni.

DayStart

Passaggio notte → giorno.

NightStart

Passaggio giorno → notte.

Allarmi meteo generati

TempLow

Temperatura < soglia bassa.

TempHigh

Temperatura > soglia alta.

WindHigh

Vento > soglia vento forte.

RainHigh

Pioggia > soglia pioggia intensa.

Gli allarmi sono inviati al frontend come array:

```
WeatherAlarm active[4];  
alarmCallback(active, count);
```

Integrazione con automazioni

Il frontend usa gli eventi meteo per attivare scene:

Esempi reali

- RainStart → WindowClose
- WindGustStart → WindowClose
- NightStart → WindowClose
- RainHigh → WindowClose

Queste scene sono definite in **AUTOMATION_JSON**, non nel firmware.

Diagramma del flusso meteo

```
Task_Meteo  
  |  
  ▼  
WeatherEngine::Loop()  
  |  
  ▼  
WeatherOrchestrator::Loop()  
  |  
  ▼  
WeatherStation.update()  
  ├── filtri 0(1)  
  ├── giorno/notte  
  ├── pioggia  
  ├── raffiche  
  ├── allarmi  
  └── callback eventi/allarmi
```

Esempio di automazione reale

```
"scenes": [  
  {  
    "name": "WindowClose",  
    "actions": [  
      { "area": AREA_D01A, "value": 0 },  
      { "area": AREA_D02A, "value": 0 }  
    ]  
  }  
]
```

Il frontend la attiva con:

```
manager.getAutomation().triggerScene("WindowClose", now);
```

Conclusione

Il modulo meteo di DomoManager è:

- robusto
- filtrato
- non bloccante
- industriale
- configurabile
- ricco di eventi
- ricco di allarmi
- integrabile con automazioni

È uno dei moduli più completi dell'intero sistema, progettato per funzionare in modo affidabile su Opta e hardware embedded.

PowerSupervisorEngine

(Analisi completa basata su PowerSupervisorOrchestrator + PowerSupervisor di DMPower.hpp)

Obiettivo del modulo

Il **PowerSupervisorEngine** è il modulo del Frontend dedicato alla **supervisione elettrica di sicurezza**. Non ha nulla a che vedere con la gestione dei carichi (che appartiene al PowerEngine), né con l'ottimizzazione energetica: **qui si parla di protezione, fault detection e monitoraggio alimentazioni critiche**.

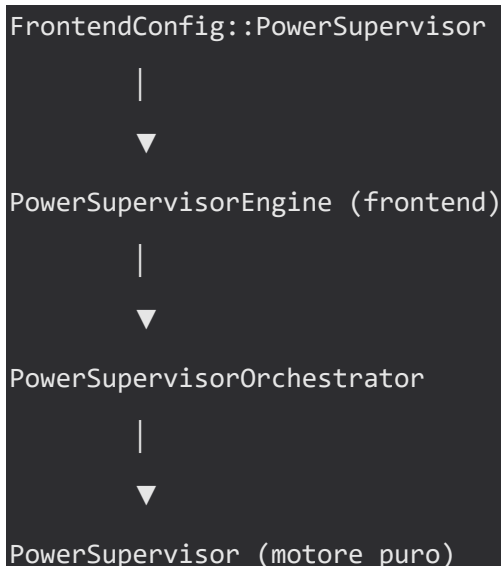
Il suo compito è:

- leggere la tensione di rete (es. 230 V)
- verificare la presenza dei 24 V DC
- rilevare fault elettrici
- rilevare modalità batteria/UPS
- generare allarmi affidabili e non ridondanti
- notificare il frontend tramite callback
- esporre segnali extra configurabili

È un modulo **deterministico, non bloccante, industriale**, progettato per funzionare anche in condizioni di errore parziale.

Architettura del modulo

Il modulo è composto da due livelli:



PowerSupervisorEngine (frontend)

- registra la callback di allarme
- aggiunge eventuali extra-signals (es. 12 V)
- chiama l'orchestratore nel loop

PowerSupervisorOrchestrator

- costruisce la configurazione dinamica

- collega le funzioni di lettura al buffer
- istanzia il motore PowerSupervisor
- inoltra gli allarmi al frontend

PowerSupervisor (motore puro)

- legge i segnali
- applica soglie
- rileva transizioni
- genera allarmi solo quando necessario
- gestisce extra-signals

Configurazione nel FrontendConfig

La sezione dedicata è:

```
struct PowerSupervisor {
    bool enabled;
    unsigned long intervalMs;

    GenericSensor::Config mainPower;
    GenericSensor::Config i12vOk;
    GenericSensor::Config i24vOk;
    GenericSensor::Config fault;
    GenericSensor::Config battery;

    float mainPowerLow;
    float mainPowerHigh;
} ps;
```

Significato dei campi

Campo	Descrizione
enabled	abilita il modulo
intervalMs	frequenza di campionamento
mainPower	lettura tensione rete (buffer/analogico)
i12vOk	lettura presenza 12 V
i24vOk	lettura presenza 24 V
fault	segnale fault elettrico
battery	segnale modalità batteria/UPS

mainPowerLow	soglia bassa tensione
mainPowerHigh	soglia alta tensione

Setup del modulo

Il frontend chiama:

```
PowerSupervisorOrchestrator::Setup(cfg, onAlarm);
```

Dove onAlarm è una callback definita dal frontend:

```
static void onAlarm(const String& name, long value) {
    LOG_WF("PowerSupervisorEngine", "[ALARM] %s = %ld", name.c_str(), value);
}
```

Allarmi disponibili

- **MainPower** → tensione fuori range
- **24V_OK** → perdita alimentazione 24 V
- **Fault** → fault elettrico generico
- **BatteryMode** → ingresso in modalità batteria/UPS
- **Extra signals** → segnali aggiuntivi configurati dal frontend

Lettura dei segnali dal buffer

L'orchestratore collega i GenericSensor::Config al buffer:

```
float v = GenericSensor::read(cfg.mainPower, buf);
bool p24 = GenericSensor::read(cfg.i24vOk, buf) != 0;
bool f = GenericSensor::read(cfg.fault, buf) != 0;
bool b = GenericSensor::read(cfg.battery, buf) != 0;
```

Questo permette di:

- leggere da buffer
- leggere da pin digitali
- leggere da analogici
- usare costanti

senza cambiare il codice del motore.

Logica di supervisione (motore puro)

Il motore PowerSupervisor implementa:

1. Supervisione tensione principale

```
if (pv.value < cfg.mainPowerLow || pv.value > cfg.mainPowerHigh)
    triggerAlarm("MainPower", pv.value);
```

2. Supervisione 24 V

Allarme quando passa da **true** → **false**.

3. Supervisione fault

Allarme quando passa da **false** → **true**.

4. Supervisione modalità batteria

Allarme quando passa da **false** → **true**.

5. Extra signals

Ogni extra-signal è definito da:

```
addExtraSignal("12V", [](){ ... });
```

Il motore genera un allarme **solo quando il valore cambia**.

Extra signals (estensione potente)

Il frontend può aggiungere segnali arbitrari:

```
PowerSupervisorOrchestrator::Get().addExtraSignal("12V", []() -> long {  
    return GenericSensor::read(cfg12v, buf) != 0 ? 1L : 0L;  
});
```

Esempi di utilizzo:

- monitoraggio alimentatore 12 V
- monitoraggio stato inverter
- monitoraggio stato UPS
- monitoraggio stato relè critici

Loop non bloccante

Il TaskEngine chiama:

```
PowerSupervisorOrchestrator::Loop(now);
```

Il motore:

- non blocca
- non usa delay
- non alloca memoria
- non genera allarmi duplicati

Integrazione con automazioni

Il frontend può reagire agli allarmi:

Esempi reali

- MainPower fuori range → spegni carichi non essenziali
- 24V_OK perso → attiva scena di emergenza
- Fault → notifica MQTT
- BatteryMode → attiva modalità risparmio

Esempio:

```
if (name == "MainPower" && value < 210) {  
    manager.getAutomation().triggerScene("EmergencyShutdown", now);  
}
```

Diagnostica

Il motore espone una diagnostica completa:

```
===== POWER SUPERVISOR =====  
MainPower: 228.5 V (VALID)  
24V OK: YES  
Fault: NO  
Battery Mode: NO  
Extra signals:  
  - 12V = 1  
===== END POWER SUPERVISOR =====
```

Diagramma del flusso

```
Task_PowerSupervisor  
  |  
  ▼  
PowerSupervisorOrchestrator::Loop()  
  |  
  ▼  
PowerSupervisor::loop()  
  ├── Legge mainPower  
  ├── Verifica soglie  
  ├── Rileva fault  
  ├── Rileva 24V  
  ├── Rileva battery mode  
  ├── Rileva extra signals  
  └─ triggerAlarm(name, value)
```

Quando usare PowerSupervisorEngine

Obbligatorio in:

- impianti con UPS

- impianti con alimentatori 24 V critici
- impianti con inverter FV
- impianti con carichi di sicurezza
- quadri elettrici industriali

Consigliato in:

- impianti residenziali con automazioni importanti
- sistemi con alimentazioni multiple

Conclusione del capitolo

Il **PowerSupervisorEngine** è un modulo fondamentale per la sicurezza elettrica del sistema:

- affidabile
- deterministico
- configurabile
- estensibile
- integrabile con automazioni

È progettato per funzionare in modo robusto anche in condizioni di errore, garantendo che il sistema possa reagire rapidamente a problemi elettrici.

PowerEngine

(Gestione energetica avanzata: FV, batteria, carichi, limiti, auto-tuning)

Obiettivo del modulo

Il **PowerEngine** è il modulo del Frontend che orchestra il motore energetico completo di DomoManager, basato su:

- **PowerManager** (orchestratore centrale)
- **PVStringManager** (fotovoltaico)
- **BatteryManager** (batteria)
- **LoadManager** (carichi normali + carichi termici)
- **LimitManager** (limiti di rete soft/hard)
- **AutoTuneManager** (auto-ottimizzazione dinamica)

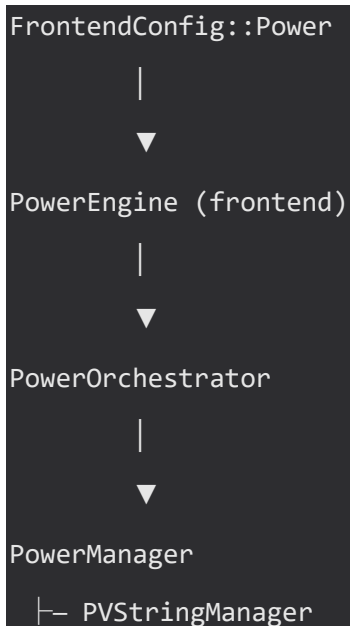
Il suo scopo è:

- ottimizzare l'autoconsumo
- proteggere il limite di potenza del contatore
- gestire carichi intelligenti
- prevedere la produzione FV
- compensare con la batteria
- generare suggerimenti (attach/stacca)
- reagire a condizioni critiche

È un modulo **industriale, modulare, deterministico**, progettato per funzionare su Opta in tempo reale.

Architettura del modulo

Il flusso è:



```
└─ BatteryManager
└─ LoadManager
└─ LimitManager
└─ AutoTuneManager
```

PowerEngine (frontend)

- registra callback (load change, warning, exceeded, error, suggestion)
- passa i parametri runtime (gridPower, lux, tempExt, produzione FV, medie, orario)
- chiama il loop del PowerOrchestrator

PowerOrchestrator

- costruisce il PowerManager
- configura limiti, carichi, thermal loads

collega callback

- esegue il ciclo completo

PowerManager

- calcola net power
- aggiorna FV, batteria, forecast
- aggiorna carichi
- applica limiti
- applica auto-tuning

8.3. Configurazione nel FrontendConfig

La sezione Power è:

```
struct Power {
    bool enabled;
    uint32_t intervalMs;

    float limitSoft;
    float limitHard;
    bool autoTune;
    uint32_t tuneIntervalMs;

    struct Load { ... } loads[8];
    size_t loadCount;
```

```
struct Thermal { ... } thermal;
} power;
```

Parametri principali

Campo	Descrizione
limitSoft	limite di potenza dinamico (soft)
limitHard	limite di sicurezza (hard)
autoTune	abilita auto-ottimizzazione
tuneIntervalMs	intervallo tuning
loads[]	carichi normali
thermal	carico termico (es. boiler, pompa di calore)

Setup del modulo

Il frontend chiama:

```
PowerOrchestrator::Setup(cfg, 3000, 3500, onLoadChange, onWarn, onExceeded,
onErr, onSug);
```

Callback disponibili

- **onLoadChange(name, state)**
- **onLimitWarning(net, limit)**
- **onLimitExceeded(net, limit)**
- **onPowerError(code, msg)**
- **onSuggestion(suggestion, severity, reason)**

Il frontend può:

- loggare
- inviare MQTT
- attivare scene
- scrivere nel buffer

Flusso operativo del PowerEngine

Il TaskEngine chiama:

```
PowerEngine::Loop(now, gridPower, lux, tempExt, actualProduction,
meanTemperature, month, hour, minute);
```

Il PowerManager esegue:

1. **Aggiornamento potenza di rete**
2. **Aggiornamento FV**
3. **Forecast FV**
4. **Aggiornamento batteria**

5. **Calcolo net power con batteria**
6. **Auto-tuning (se attivo)**
7. **Aggiornamento carichi termici**
8. **Aggiornamento carichi normali**
9. **Verifica limiti**
10. **Aggiornamento diagnostica**

PVStringManager — gestione fotovoltaico

Il modulo FV gestisce:

- più stringhe indipendenti
- potenza reale
- forecast basato su:
 - alba/tramonto
 - curva sinusoidale
 - lux
 - temperatura esterna
 - fattori di stringa

Forecast multi-stringa

```
theoretical = nominalWatt * sin(π * progress)
lightFactor = min(lux / 100000, 1)
tempFactor  = 1 - 0.0045 * (T - 25)
forecast = theoretical * lightFactor * tempFactor
```

Volatilità lux

Usata per auto-tuning.

BatteryManager — gestione batteria

Il modulo gestisce:

- SOC
- SOH
- temperatura
- potenza di carica/scarica
- protezioni:
 - SOC minimo/massimo
 - temperatura minima/massima

Logica principale

- surplus FV → carica

- deficit → scarica
- forecast alto → carica anticipata
- anti-oscillazione → 30s min tra cicli

LoadManager — gestione carichi

Gestisce due categorie:

1. Carichi normali

Con:

- minOn
- minOff
- priorità
- suggerimenti attach/stacca
- hysteresis dinamica

2. Carichi termici

Con:

- baseTarget
- comfortMin/Max
- boost FV
- suggerimenti termici

Suggerimenti

Il modulo non attiva carichi: **genera suggerimenti**, che il frontend può interpretare.

Esempi:

- "attacca:lavatrice"
- "stacca:boiler"

LimitManager — protezione limiti

Gestisce:

- limite soft
- limite hard
- warning threshold
- callback dedicate

Logica

```
if net >= limit * warningPct → warning
if net >= limit → exceeded
if net >= hardLimit → hardLimitCallback
```

AutoTuneManager — auto-ottimizzazione

Regola dinamicamente:

- hysteresis ON/OFF
- minOn/minOff

Basato su:

- volatilità lux
- forecast error
- cicli carichi
- stato batteria

Obiettivo

Ridurre oscillazioni e migliorare autoconsumo.

Esempio reale di ciclo Power

Supponiamo:

- gridPower = 2800 W
- limitSoft = 3000 W
- FV = 500 W
- batteria = 0
- net = 2300 W

Il PowerManager:

1. net < limitSoft → nessun detach
2. forecast alto → suggerisce attach carichi termici
3. lux volatile → aumenta hysteresis
4. nessun warning

Integrazione con automazioni

Esempi:

Superamento limite

Codice

```
onLimitExceeded → triggerScene("LoadShed")
```

Suggerimento carico

Codice

```
onSuggestion("attacca:boiler") → scrivi area D0
```

Errore inverter

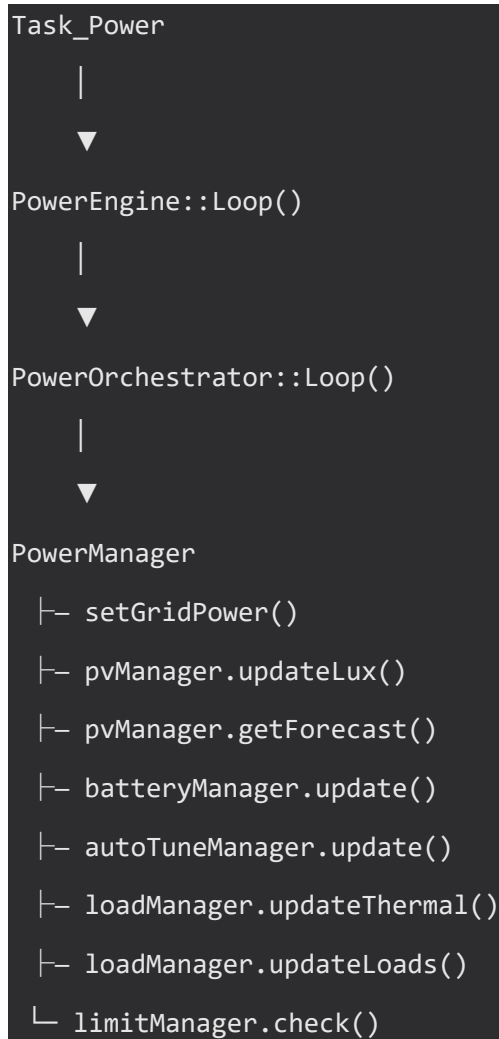
```
onPowerError → notifica MQTT
```

Diagnostica

Il **PowerManager** espone:

- potenza FV
- potenza batteria
- forecast
- net power
- stato carichi
- stato thermal loads
- hysteresis
- forecast error
- volatilità lux

Diagramma del flusso PowerEngine



Conclusione del capitolo

Il **PowerEngine** è uno dei moduli più avanzati dell'intero DomoManager:

- modulare

- predittivo
- auto-ottimizzante
- integrato con FV e batteria
- sicuro
- configurabile
- industriale

Permette di trasformare un impianto Opta in un vero **energy manager** professionale.

SecuritySensorEngine

(Analisi completa basata su SecuritySensorEngine, SecurityOrchestrator, WiredSensorsManager, ZoneManager, AlarmDispatcher)

Questo capitolo è uno dei più complessi e importanti dell'intero onboarding avanzato, perché descrive **il motore di sicurezza cablata** di DomoManager: un sistema industriale, deterministico, modulare e completamente configurabile via DMSetup.hpp.

Obiettivo del modulo

Il **SecuritySensorEngine** è il modulo del Frontend che:

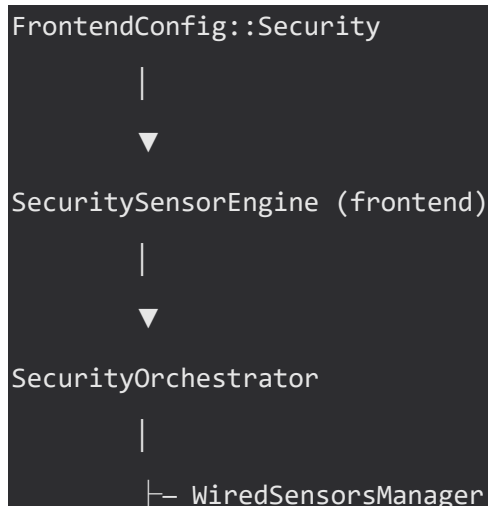
- inizializza il motore di sicurezza (SecurityOrchestrator)
- registra callback dinamiche per zone, tipi e combinazioni
- esegue il ciclo di lettura sensori
- aggiorna il SystemManager (bitmask sicurezza)
- scrive nel buffer l'aggregato sicurezza
- espone API statiche per comandi e diagnostica

Il motore gestisce:

- PIR (RT / H24)
- sensori finestra
- sensori porta
- sensori allagamento
- sensori fumo
- tamper
- sensori custom

È progettato per funzionare su Opta in modo **deterministico, non bloccante, senza allocazioni ricorrenti**, con **debounce, timer TON, memorie, startup inhibit, zone dinamiche, callback dinamiche**.

Architettura del modulo



```

|      |-- Sensor
|      |-- SensorChannel
|      |-- ZoneManager
|      |-- AlarmDispatcher
|
|-- SystemManager (bitmask)

```

Configurazione nel FrontendConfig

```

struct Security {
    bool enabled;

    uint32_t intervalMs;

    const WiredSensorsManager::WiredSensorConfig* sensors;

    size_t count;

    uint32_t startupInhibitMs;
} security;

```

WiredSensorConfig

```

struct WiredSensorConfig {
    const char* zone;

    std::initializer_list<SensorChannel> channels;

    std::initializer_list<std::function<bool()>> readers;

    SensorCategory category;

    int cmdArea;
};

```

Esempio reale (DMSetup.hpp)

```

{
    "ZonaIngresso",
    { { -1, RT_DELAY, SensorChannelType::RT } },
    { [](){ return digitalRead(5); } },
    SensorCategory::PIR,
    -1
}

```

Il cuore del sistema: WiredSensorsManager

Il WiredSensorsManager è il motore che:

- crea i sensori
- li assegna alle zone
- gestisce startup inhibit
- legge i valori tramite funzioni configurate
- aggiorna timer, debounce, memorie
- calcola lo stato aggregato (intrusione, flood, smoke, tamper...)

SensorChannel — il singolo canale

Ogni canale ha:

- pin
- TON timer
- FastDebounce
- mem (memoria allarme)
- inhibit
- SensorChannelType (RT, H24, LEN, MASK)

Tipi di canale

Tipo	Significato
RT	Ritardato (timer TON)
H24	Sorveglianza continua
LEN	Lente (es. contatti magnetici)
MASK	Tamper / mascheramento

Sensor — aggregazione canali

Ogni sensore:

- contiene più canali
- ha una lookup per tipo
- ha startupInhibit
- ha memorie per allarmi persistenti
- produce alarmOut

Logica principale

```
if (timer.Q() && !inhibit)
    tempAlarm = true;

if (_engage)
    ch.mem = true;
```

Startup inhibit

Durante i primi X ms:

- nessun allarme
- nessuna memoria
- timer fermi

ZoneManager — gestione zone

Il ZoneManager gestisce:

- sensori per zona
- allarmi per zona
- allarmi per tipo
- snooze
- reset
- enable/disable
- engage/disengage
- dispatch eventi

Funzioni chiave

- ZoneAlarm(zone)
- ZoneAlarmByType(zone, type)
- NewAlarmByType(type)
- Snooze(type)
- EnableZone(zone, mode)
- EngageZone(zone, mode)

AlarmDispatcher — callback dinamiche

Il sistema supporta 4 livelli di callback:

1. **Globali** OnAnyAlarm(cb)
2. **Per tipo** OnAlarmType(RT, cb)
3. **Per zona** OnZoneAlarm("Ingresso", cb)
4. **Per zona + tipo** OnZoneAlarmType("Ingresso", RT, cb)

Il frontend registra dinamicamente tutte le callback in SecuritySensorEngine::registerSecurityCallbacks().

SecurityOrchestrator — orchestratore backend

Il SecurityOrchestrator:

- valida la configurazione
- inizializza i sensori
- applica startup inhibit globale
- abilita ed engage tutti i sensori

- esegue il loop
- aggiorna il SystemManager
- rileva cambiamenti
- stampa diagnostica

SystemManager — bitmask sicurezza

Il SystemManager aggrega:

Campo	Significato
ALARM_INTRUSION	PIR RT
ALARM_INTRUSION_H24	PIR H24
ALARM_FLOOD	allagamento
ALARM_SMOKE	fumo
WINDOWS_OPEN	finestre aperte
DOORS_OPEN	porte aperte
ALARM_TAMPER	tamper

Bitmask finale

```
int mask = system.getBitmask();
```

Scritta nel buffer dal TaskEngine:

```
buffer.WriteElement(22, Field, mask, now);
```

SecuritySensorEngine (frontend)

Il frontend:

- chiama SecurityOrchestrator::Setup()
- registra callback dinamiche
- esegue il loop
- espone accessor statici

Callback registrate

- global
- per tipo
- per zona
- per zona+tipo

Esempio:

```
SecurityOrchestrator::RegisterCallbackZone("Ingresso", onZoneAlarm);
```

Loop del SecuritySensorEngine

```
bool changed = SecurityOrchestrator::Loop(now);
```

Se changed == true:

- stampa diagnostica zone
- stampa diagnostica SystemManager
- TaskEngine scrive bitmask nel buffer

Integrazione con automazioni

Esempi:

Intrusione

```
if (mask & (1 << ALARM_INTRUSION))
    triggerScene("AllarmeIntrusione");
```

Finestra aperta

```
if (mask & (1 << WINDOWS_OPEN))
    triggerScene("BloccaClima");
```

Allagamento

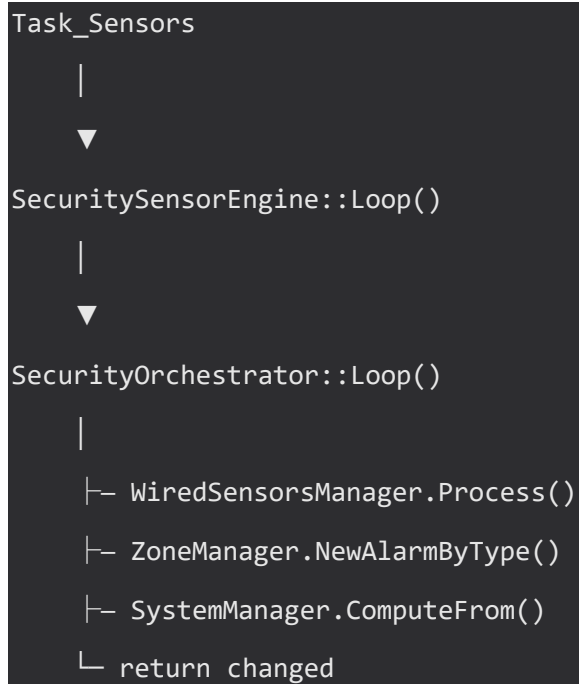
```
if (mask & (1 << ALARM_FLOOD))
    triggerScene("ChiudiElettrovalvola");
```

Diagnostica

Il motore stampa:

```
===== SECURITY ZONES =====
Ingresso → ALARM
Cucina → OK
Camera → OK
===== END SECURITY ZONES =====
Intrusion: TRUE
Flood: FALSE
Smoke: FALSE
Windows Open: FALSE
Doors Open: FALSE
Tamper: FALSE
Bitmask: 1
```

Diagramma del flusso



Conclusione

Il **SecuritySensorEngine** è un motore di sicurezza industriale:

- completamente configurabile
- deterministico
- con debounce, timer, memorie
- con zone dinamiche
- con callback multilivello
- con bitmask aggregata
- integrabile con automazioni

JobsEngine

(Motore asincrono per sequenze, workflow e macro non bloccanti)

Obiettivo del modulo

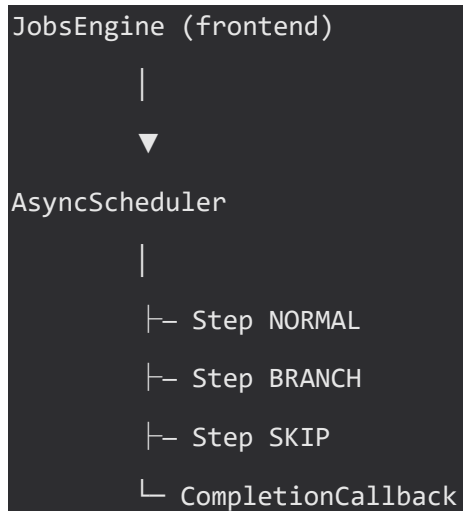
Il **JobsEngine** è il modulo del Frontend che:

- costruisce job complessi composti da step sequenziali
- integra condizioni, branch, skip dinamici
- esegue operazioni lunghe senza bloccare il ciclo principale
- permette di definire “macro” di automazione avanzate
- sfrutta il motore asincrono **AsyncScheduler**
- permette di creare workflow deterministici, ripetibili e sicuri

È un modulo pensato per:

- sequenze HVAC (es. pre-riscaldamento, cicli ACS)
- macro di arrivo/uscita casa
- cicli di ventilazione
- sequenze di luci
- operazioni lente (es. attese, temporizzazioni, controlli)

Architettura del modulo



Il JobsEngine:

- crea i job
- li registra nello scheduler
- definisce le funzioni degli step
- definisce condizioni e skip
- esegue il loop dello scheduler

AsyncScheduler — il cuore del sistema

La classe AsyncScheduler è un **motore asincrono deterministico**, progettato per:

- non bloccare mai
- eseguire job multipli in parallelo
- rispettare priorità
- gestire delay tra step
- gestire condizioni e branch
- gestire skip dinamici
- eseguire callback di completamento

Struttura di uno Step

```
struct Step {  
    StepType type;           // NORMAL_STEP o BRANCH_STEP  
    FunctionPointer fnc;     // funzione dello step  
    unsigned long delayAfterMs; // delay dopo completamento  
    ConditionFunction condition; // per branch  
    int thenStep;  
    int elseStep;  
    ConditionFunction skipIf; // condizione di skip  
    String description;  
};
```

Struttura di un Job

```
struct Job {  
    std::vector<Step> steps;  
    int currentStep;  
    bool active;  
    bool cancelled;  
    unsigned long nextRunTime;  
    int priority;  
    CompletionCallback onComplete;  
    uint8_t priorityWeight;  
    uint8_t counter;  
    String name;  
};
```

Tipologie di Step

1. NORMAL_STEP

Esegue una funzione:

```
bool done = step.fnc(context);
```

- se done == true → passa allo step successivo
- se done == false → riprova al prossimo ciclo

2. BRANCH_STEP

Esegue una condizione:

```
if (condition(ctx)) goto thenStep;  
else goto elseStep;
```

3. SKIP

Se la condizione è vera:

```
if (skipIf(ctx)) salta lo step
```

Priorità dei job

Ogni job ha:

- priority → ordine di esecuzione
- priorityWeight → quanti cicli deve attendere prima di rieseguire

Questo permette di:

- dare priorità a job critici
- rallentare job meno importanti
- evitare starvation

JobsEngine — implementazione frontend

Il JobsEngine contiene:

- funzioni statiche che rappresentano gli step
- condizioni
- callback di completamento
- costruzione dei job
- registrazione nello scheduler

Esempio reale (dal tuo codice)

Funzione di raffreddamento

```
static bool coolDown(void* ctx) {  
    LOG_IF("Scheduler", "Cool down..");  
    return true;  
}
```

Funzione di riscaldamento (lunga)

```
static bool heatUp(void* ctx) {
    static unsigned long start = 0;
    unsigned long now = millis();

    if (start == 0) {
        start = now;
        LOG_IF("Scheduler", "Heating started");
    }

    if (now - start >= 50000) {
        LOG_IF("Scheduler", "Heating finished");
        start = 0;
        return true;
    }
    return false;
}
```

Funzione lunga generica

```
static bool function3(void* ctx) {
    static unsigned long start = 0;
    unsigned long now = millis();

    if (start == 0) {
        start = now;
        LOG_IF("Scheduler", "Long task started");
    }

    if (now - start >= 5000) {
        LOG_IF("Scheduler", "Long task finished");
        start = 0;
        return true;
    }
}
```

```
    return false;
}
```

10.7. Condizioni e skip

Condizione “isHot”

```
static bool isHot(void* ctx) {
    auto& buf = DomoManager::instance->getBuffer();
    BufferSourceInfo info;
    if (buf.GetData(33, Field, info))
        return info.value > 35;
    return analogRead(A0) > 600;
}
```

Skip esterno

```
static bool myExternalSkipCondition(void* ctx) {
    return true;
}
```

Costruzione dei job

Job 0 — “Quando arrivo a casa”

1. Step BRANCH → verifica temperatura
2. Step NORMAL → coolDown
3. Step NORMAL → heatUp

Job 1 — “Avvio riscaldamento”

1. Step NORMAL → heatUp
2. Step NORMAL → function3 (con skip)

Esecuzione del JobsEngine

Il TaskEngine chiama:

```
JobsEngine::Loop(now);
```

Che esegue:

```
scheduler.run(now);
```

Integrazione con automazioni

I job possono:

- attivare scene
- scrivere nel buffer
- comandare dispositivi

- interagire con HVAC, Power, Security

Esempi:

Sequenza di arrivo a casa

- accendi luci
- avvia riscaldamento
- apri vasistas
- attiva profilo comfort

Sequenza di uscita

- spegni luci
- chiudi finestre
- attiva antifurto
- abbassa carichi

Esempio di job avanzato (consigliato)

Workflow:

```
1. Verifica finestra aperta
2. Se aperta → chiudi
3. Attendi 5 secondi
4. Avvia riscaldamento
5. Attendi temperatura target
6. Notifica completamento
```

Diagnostica

Lo scheduler espone:

- lista job
- stato
- step corrente
- priorità
- ritardi
- cancellazioni

Puoi aggiungere:

```
for (auto& j : scheduler.getJobs()) {
    LOG("JOB", "%s step=%d active=%d", j.name.c_str(), j.currentStep, j.active);
}
```

Conclusione del capitolo

Il **JobsEngine** è un modulo potentissimo:

- asincrono
- deterministico
- non bloccante
- modulare
- configurabile
- integrabile con tutti i motori del sistema

Permette di creare **workflow complessi** senza mai bloccare il ciclo principale, mantenendo la stabilità industriale del firmware.

TaskEngine

(Il cuore del ciclo deterministico del Frontend)

Obiettivo del modulo

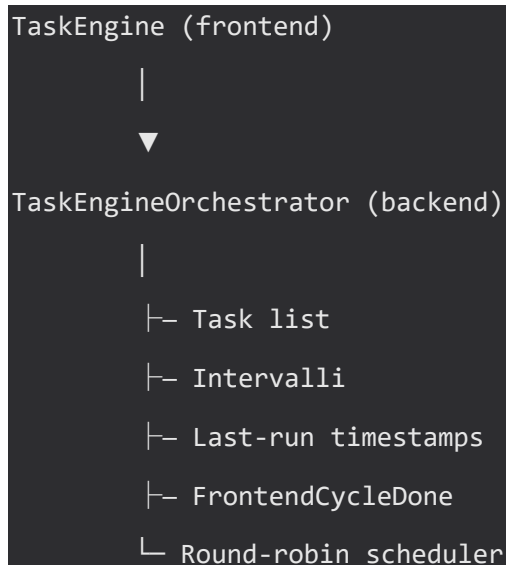
Il **TaskEngine** è il modulo che garantisce:

- esecuzione **non bloccante** di tutti i motori
- **rate-limiting** preciso per ogni task
- **round-robin deterministico**
- nessuna attesa attiva
- nessun delay()
- nessun jitter critico
- priorità implicite basate sugli intervalli
- sincronizzazione con il ciclo del DomoManager

È il componente che permette a Opta di comportarsi come un **PLC soft-real-time**, pur mantenendo la flessibilità di un firmware embedded.

Architettura del TaskEngine

Il TaskEngine è composto da due livelli:



Ogni task è definito da:

- una funzione
- un intervallo minimo
- un timestamp dell'ultima esecuzione

Perché il TaskEngine è fondamentale

Senza TaskEngine:

- HVAC bloccherebbe il loop

- Meteo farebbe letture troppo frequenti
- PowerEngine consumerebbe tutta la CPU
- Security genererebbe jitter
- JobsEngine non sarebbe deterministico
- WebAPI e MQTT saturerebbero la rete
- DomoManager non avrebbe un ciclo stabile

Con TaskEngine:

- ogni modulo gira **solo quando serve**
- nessun modulo blocca gli altri
- il sistema rimane fluido anche sotto carico
- la diagnostica è coerente
- il comportamento è ripetibile

Registrazione dei task

Il Frontend registra i task nel TaskEngineOrchestrator:

Esempio (semplificato):

```
TaskEngineOrchestrator::AddTask("HVAC", cfg.hvac.intervalMs, Task_HVAC);
TaskEngineOrchestrator::AddTask("Weather", cfg.weather.intervalMs,
Task_Weather);
TaskEngineOrchestrator::AddTask("Power", cfg.power.intervalMs, Task_Power);
TaskEngineOrchestrator::AddTask("Security", cfg.security.intervalMs,
Task_Security);
TaskEngineOrchestrator::AddTask("Jobs", cfg.jobs.intervalMs, Task_Jobs);
```

Ogni task ha:

- nome
- intervallo
- funzione da eseguire

Esecuzione del ciclo

Il main chiama:

```
TaskEngineOrchestrator::Loop(now);
```

Il loop:

1. scorre tutti i task
2. verifica se l'intervallo è scaduto
3. se sì → esegue il task
4. aggiorna il timestamp

5. passa al task successivo

Round-robin puro

Non esiste priorità esplicita: **la priorità è determinata dall'intervallo.**

FrontendCycleDone

Il TaskEngine gestisce un flag speciale:

```
frontendCycleDone
```

Serve per:

- sincronizzare DomoManager
- evitare race condition
- garantire che ogni ciclo sia completo
- permettere al Bridge di sapere quando inviare pacchetti
- permettere al WebAPIEngine di sapere quando è sicuro inviare richieste

Tipologie di task

1. Task di dominio

- HVAC
- Weather
- Power
- Security
- Averages
- Jobs

2. Task di comunicazione

- MQTT
- WebAPI
- Bridge

3. Task di sistema

- DomoManager::Loop
- diagnostica
- watchdog

Esempio reale di task

Task HVAC

```
void Task_HVAC(unsigned long now) {  
    HVACEngine::Loop(now, zoneTemps, tInt, tExt, windowOpen, hvacTime);  
}
```

Task Security

```
void Task_Security(unsigned long now) {
    bool changed = SecurityOrchestrator::Loop(now);
    if (changed) writeBitmaskToBuffer();
}
```

Task Power

```
void Task_Power(unsigned long now) {
    PowerOrchestrator::Loop(now, gridPower, lux, tempExt, pv, meanTemp, month,
hour, minute);
}
```

Determinismo del ciclo

Il TaskEngine garantisce:

- nessun task può bloccare il ciclo
- nessun task può monopolizzare la CPU
- ogni task ha un intervallo minimo
- ogni task viene eseguito al massimo una volta per intervallo
- l'ordine dei task è fisso

Questo è fondamentale per:

- sicurezza
- affidabilità
- diagnostica coerente
- comportamento prevedibile

Integrazione con JobsEngine

Il JobsEngine viene eseguito come un task:

```
TaskEngineOrchestrator::AddTask("Jobs", cfg.jobs.intervalMs, Task_Jobs);
```

Questo garantisce:

- esecuzione asincrona
- nessun blocco
- priorità controllata
- workflow deterministici

Integrazione con automazioni

Le automazioni JSON vengono eseguite tramite task:

- scene
- rules
- composite

- autoOff
- windowCloser
- sequences

Ogni automazione è un task non bloccante.

Diagramma del flusso TaskEngine

```
main loop
|
▼
TaskEngineOrchestrator::Loop()
|
├─ Task_HVAC (ogni X ms)
├─ Task_Weather (ogni Y ms)
├─ Task_Power (ogni Z ms)
├─ Task_Security (ogni N ms)
├─ Task_Averages
├─ Task_Jobs
├─ Task_MQTT
├─ Task_WebAPI
└─ Task_DomoManager
```

Conclusione

Il **TaskEngine** è il cuore del sistema:

- mantiene tutto fluido
- evita blocchi
- garantisce determinismo
- permette di scalare a decine di moduli
- rende Opta un vero controllore industriale

Senza TaskEngine, DomoManager non sarebbe affidabile. Con TaskEngine, diventa una piattaforma solida, modulare e professionale.

AutomationEngine (AUTOMATION_JSON)

(Motore di automazione dichiarativa basato su JSON, scene, regole, sequenze e logiche composite)

Obiettivo del modulo

L'**AutomationEngine** è il motore che esegue tutte le automazioni definite nel file:

```
AUTOMATION_JSON
```

Il suo scopo è:

- interpretare scene
- valutare condizioni
- eseguire regole dinamiche
- gestire regole temporali
- gestire sequenze temporizzate
- combinare logiche complesse (trend, debounce, bitmask, TOF)
- scrivere valori nel buffer
- orchestrare automazioni in modo deterministico e non bloccante

È il **motore logico** del sistema, completamente configurabile via JSON.

Architettura generale

```
AUTOMATION_JSON
|
▼
AutomationBuilder
|
▼
AutomationEngine
|
├─ Scene
├─ Rule
├─ ScheduledRule
├─ DynamicAutomation
├─ ActionSequence
└─ CompositeRule
```

AutomationBuilder

- interpreta il JSON
- costruisce scene, regole, sequenze

- crea oggetti dinamici (TrendRule, ThresholdRule, CompositeRule...)
- registra tutto nell'AutomationEngine

AutomationEngine

- esegue automazioni
- valuta condizioni
- attiva scene
- aggiorna sequenze
- gestisce regole temporali
- esegue dynamic automations

Scene — blocchi di azioni

Una **Scene** è un insieme di azioni:

```
{
  "name": "WindowClose",
  "actions": [
    { "area": "AREA_WIN1_CMD", "value": 0 },
    { "area": "AREA_WIN2_CMD", "value": 0 }
  ]
}
```

Ogni scena:

- contiene fino a 16 azioni
- scrive nel buffer
- evita scritture ridondanti
- forza l'esecuzione allo startup
- traccia gli ultimi valori scritti (per edge detection)

Esecuzione scena

```
engine->triggerScene("WindowClose", now);
```

Rule — regole classiche

Una regola è composta da:

- condizioni (AND/OR)
- azioni THEN
- azioni ELSE
- memoria dell'ultimo risultato
- edge detection

Condizioni supportate

- >
- <
- >=
- <=
- ==
- !=
- BETWEEN

Esempio JSON

```
{
  "name": "HighTemp",
  "type": "threshold",
  "intervalMs": 2000,
  "threshold": { "area": "TEMP", "op": ">", "threshold": 27 },
  "sceneTrue": "Cooling",
  "sceneFalse": "Idle"
}
```

ScheduledRule — regole temporali

Tipi supportati:

Tipo	Descrizione
EVERY_INTERVAL	esecuzione periodica
EVERY_DAY_AT	orario preciso (HH:MM)
SEQUENCE_AT_TIME	avvia sequenza a orario
SEQUENCE_EVERY_INTERVAL	sequenza periodica
SEQUENCE_WITH_TIMEOUT	sequenza con timeout + fallback

Esempio: esecuzione giornaliera

```
{
  "name": "NightMode",
  "type": "time",
  "threshold": { "threshold": 22*60 },
  "sceneTrue": "Scene_Night"
}
```

DynamicAutomation — logiche avanzate

Le **dynamic automations** sono callback periodiche che implementano logiche complesse.

Sono create automaticamente dal Builder per:

- TrendRule
- ThresholdRule
- BitmaskRule
- MultiRule
- TimeRule
- SequenceRule
- CompositeRule

Esempio: trend dinamico

```
{
  "name": "HumidityTrend",
  "type": "trend",
  "intervalMs": 5000,
  "trend": {
    "area": "HUM",
    "scale": 10,
    "threshold": 65,
    "trend": "rising"
  },
  "sceneTrue": "FanOn",
  "sceneFalse": "FanOff"
}
```

ActionSequence — sequenze temporizzate

Una sequenza è una lista di step:

```
{
  "intervalMs": 60000,
  "steps": [
    { "area": "LIGHT1", "value": 1, "delayMs": 0 },
    { "area": "LIGHT2", "value": 1, "delayMs": 2000 },
    { "area": "LIGHT1", "value": 0, "delayMs": 5000 }
  ]
}
```

Supporta:

- start
- update
- delay tra step
- timeout
- fallback

CompositeRule — logiche complesse multi-input

È il motore più potente del sistema.

Supporta input:

- **simple** → area != 0
- **debounce** → soglia + debounce + TOF
- **trend** → trend + soglia + TOF
- **bitmask** → bit specifico
- **TOF** → timer off delay

Esempio JSON

```
{
  "name": "KitchenExtractor",
  "type": "composite",
  "intervalMs": 20000,
  "composite": {
    "logic": "OR",
    "inputs": [
      { "type": "debounce", "area": "CURR", "threshold": 1, "debounceMs": 10000 },
      { "type": "trend", "area": "HUM", "scale": 10, "threshold": 65, "trend": "rising" },
      { "type": "bitmask", "area": "MANUAL_BITS", "bitIndex": 2 }
    ],
    "output": { "area": "EXTRACTOR_BITS", "bitIndex": 0 }
  },
  "sceneTrue": "CappaOn",
  "sceneFalse": "CappaOff"
}
```

AutomationBuilder — pipeline di costruzione

Il Builder esegue:

1. `parseJson()`

- legge scenes, rules, sequences
- risolve aree tramite AreaRegistry
- valida input

2. `buildScenes()`

- crea tutte le scene nel motore

3. `buildDynamicRules()`

- crea oggetti dinamici:
 - TrendRule
 - ThresholdRule
 - BitmaskRule
 - MultiRule
 - TimeRule
 - SequenceRule
 - CompositeRule

4. `buildSequences()`

- crea ActionSequence autonome

Ciclo di esecuzione dell'AutomationEngine

Il TaskEngine chiama:

```
engine.update(now);
```

Il motore:

1. esegue regole classiche
2. esegue regole schedulate
3. esegue dynamic automations
4. aggiorna sequenze
5. attiva scene
6. scrive nel buffer

Integrazione con gli altri moduli

Il motore di automazione è il **cervello logico** del sistema.

Interagisce con:

- HVACEngine
- WeatherEngine
- PowerEngine
- SecuritySensorEngine

- JobsEngine
- MQTT
- WebAPI

Esempi:

- pioggia → chiudi finestre
- umidità crescente → accendi estrattore
- superamento limite → stacca carichi
- presenza → accendi luci

Diagnostica

Il motore espone:

- scene caricate
- regole caricate
- sequenze
- dynamic automations
- dump completo della configurazione

AutomationBuilder::ReportAutomationConfig(cfg) stampa tutto in modo leggibile.

Conclusione del capitolo

L'**AutomationEngine** è un motore di automazione industriale:

- dichiarativo
- modulare
- estensibile
- non bloccante
- configurabile via JSON
- integrato con tutti i motori del sistema
- capace di logiche complesse senza scrivere C++

È il cuore dell'intelligenza del sistema DomoManager.

Best Practices per AUTOMATION_JSON

(Linee guida professionali per progettare automazioni robuste, scalabili e industriali)

Principio fondamentale: tutto passa dal Buffer

Le automazioni **non devono mai**:

- chiamare direttamente moduli HVAC, Power, Security, Weather
- manipolare stati interni
- accedere a logiche di basso livello

Devono **solo scrivere nel buffer**.

Questo garantisce:

- disaccoppiamento totale
- testabilità
- stabilità
- compatibilità futura

Usa SEMPRE scene, mai azioni dirette

Anche se una regola ha una sola azione, crea comunque una scena.

✗ Sbagliato

json

```
"sceneTrue": "",  
"action": { "area": "D01", "value": 1 }
```

✓ Corretto

```
"sceneTrue": "LuceOn"
```

E poi:

```
"scenes": [  
  { "name": "LuceOn", "actions": [ { "area": "D01", "value": 1 } ] }  
]
```

Motivo: Le scene sono idempotenti, tracciate, loggate, ottimizzate e integrate nel motore.

Usa “NoAction” quando non vuoi fare nulla

Il motore gestisce “NoAction” come scena speciale:

```
"sceneFalse": "NoAction"
```

Evita log inutili e comportamenti indesiderati.

Non duplicare logiche già presenti nei moduli

Esempi:

- Non rifare la logica meteo nel JSON → usa gli eventi del WeatherEngine.
- Non rifare la logica HVAC → usa scene che scrivono aree.
- Non rifare la logica Security → usa bitmask.

Il JSON deve orchestrare, non sostituire i moduli.

Usa CompositeRule per logiche complesse

Se hai più condizioni, non creare 5 regole separate.

✗ Sbagliato

- regola 1: corrente > 1
- regola 2: umidità rising
- regola 3: bit manuale
- regola 4: OR tra le precedenti

✓ Corretto

Una sola CompositeRule:

```
"composite": {
  "logic": "OR",
  "inputs": [
    { "type": "debounce", ... },
    { "type": "trend", ... },
    { "type": "bitmask", ... }
  ]
}
```

Usa TrendRule solo quando serve davvero

TrendRule è potente ma costoso:

- usa medie
- usa gruppi
- calcola trend
- mantiene storico

Usalo solo quando:

- serve rilevare una variazione (rising/falling)
- serve evitare falsi positivi
- serve una logica predittiva

Per soglie semplici → usa ThresholdRule.

15.7. Usa debounce e TOF per evitare oscillazioni

Esempio:

```
{
  "type": "debounce",
  "threshold": 1,
  "debounceMs": 10000,
  "tofMinutes": 5
}
```

Regola d'oro:

- debounceMs → evita attivazioni premature
- tofMinutes → evita spegnimenti prematuri

Usa MultiRule solo per condizioni semplici

MultiRule è perfetta per:

- combinare soglie
- combinare bitmask
- logiche AND/OR semplici

Se serve:

- trend
- debounce
- TOF
- bitmask + trend
- scaling

→ usa CompositeRule.

Usa ScheduledRule per orari, mai delay manuali

✗ Non usare sequenze con delay per simulare orari ✓ Usa:

```
"type": "time"
```

Oppure:

```
"type": "EVERY_DAY_AT"
```

Le sequenze devono essere brevi e deterministiche

Una sequenza deve:

- avere pochi step
- avere delay chiari
- non dipendere da condizioni esterne

- non essere troppo lunga (max 10 step)

Esempio corretto:

```
"steps": [
  { "area": "LUCE1", "value": 1, "delayMs": 0 },
  { "area": "LUCE2", "value": 1, "delayMs": 2000 },
  { "area": "LUCE1", "value": 0, "delayMs": 5000 }
]
```

Evita scene con troppi comandi

✗ Sbagliato: scene con 20 azioni ✓ Corretto: spezzare in scene logiche

Esempio:

- Scene_Alloff_Luci
- Scene_Alloff_Clima
- Scene_Alloff_Sicurezza

Poi una scena master:

```
Scene_Alloff
```

Che attiva le altre.

Usa nomi scena coerenti e leggibili

Esempi consigliati:

- WindowClose
- CoolingOn
- CoolingOff
- AllarmeIntrusione
- LoadShed
- NightMode

Evita:

- sc1
- azione_3
- pippo

Evita condizioni su aree non inizializzate

Il motore logga:

```
Condizione su area X ignorata: valore non inizializzato
```

Best practice:

- inizializzare tutte le aree nel buffer

- usare valori di default
- evitare condizioni su sensori non ancora letti

Evita regole con intervalli troppo brevi

✗ 50 ms ✗ 100 ms ✗ 200 ms

✓ 1000 ms ✓ 2000 ms ✓ 5000 ms

Il motore è industriale, non un PLC a 1 ms.

Usa “NoAction” per evitare log inutili

Esempio:

```
"sceneFalse": "NoAction"
```

Evita spam di log e scritture ridondanti.

Non duplicare scene o regole

Se due regole attivano la stessa scena:

✗ Sbagliato: duplicare la scena ✓ Corretto: riutilizzare la stessa scena

Usa ActionSequence per animazioni, non per logica

Le sequenze sono perfette per:

- luci scenografiche
- accensioni progressive
- cicli di ventilazione

Non usarle per:

- logica HVAC
- logica sicurezza
- logica power

Mantieni il JSON leggibile

Consigli:

- indentazione 2 spazi
- nomi chiari
- commenti esterni (non nel JSON)
- separare scene, regole, sequenze

Test incrementale

Procedura consigliata:

1. carica solo le scene
2. aggiungi una regola semplice
3. verifica log

4. aggiungi composite
5. aggiungi sequenze
6. aggiungi regole temporali

Conclusione del capitolo

Le best practice per AUTOMATION_JSON garantiscono:

- stabilità
- leggibilità
- scalabilità
- comportamento deterministico
- integrazione perfetta con HVAC, Power, Security, Weather

Seguendo queste linee guida, il tuo sistema diventa **un motore di automazione industriale professionale**, robusto e facile da mantenere.

Esempi completi di AUTOMATION_JSON

(Scene, regole, composite, sequenze, trend, threshold, bitmask, time rules)

Struttura generale del file AUTOMATION_JSON

Un file completo contiene tre sezioni principali:

```
{  
  "scenes": [ ... ],  
  "rules": [ ... ],  
  "sequences": [ ... ]  
}
```

- **scenes** → insiemi di azioni atomiche
- **rules** → logiche dinamiche (threshold, trend, composite, bitmask, multi, time, sequence)
- **sequences** → sequenze temporizzate autonome

Esempio 1 — Automazioni HVAC (Cappa Cucina + Bagno Umido)

(Trend + Debounce + Bitmask → CompositeRule)

Scene

```
{
  "scenes": [
    {
      "name": "CappaOn",
      "actions": [
        { "area": "AREA_CUCINA_ESTRATTORE_CMD", "value": 1 }
      ]
    },
    {
      "name": "CappaOff",
      "actions": [
        { "area": "AREA_CUCINA_ESTRATTORE_CMD", "value": 0 }
      ]
    },
    {
      "name": "BagnoUmido",
      "actions": [
        { "area": "AREA_BAGNO_ESTRATTORE_CMD", "value": 1 }
      ]
    },
    {
      "name": "BagnoNormale",
      "actions": [
        { "area": "AREA_BAGNO_ESTRATTORE_CMD", "value": 0 }
      ]
    }
  ]
}
```

Regole dinamiche (CompositeRule)

Cappa Cucina (OR tra 3 condizioni)

```
{
  "rules": [
    {
      "name": "CappaCucina",
      "type": "composite",
      "intervalMs": 20000,
      "composite": {
        "logic": "OR",
        "inputs": [
          {
            "type": "debounce",
            "name": "CorrenteCappa",
            "area": "SENSORE_CUCINA_CORRENTE",
            "threshold": 1,
            "debounceMs": 10000,
            "tofMinutes": 10
          },
          {
            "type": "trend",
            "name": "UmiditaCucina",
            "area": "SENSORE_CUCINA_HUM",
            "scale": 10,
            "threshold": 65,
            "trend": "rising",
            "tofMinutes": 5
          },
          {
            "type": "bitmask",
            "name": "ComandoManuale",
            "area": "AREA_CUCINA ESTRATTORE_BIT",
```

```

        "bitIndex": 0
    }
],
"output": {
    "area": "AREA_CUCINA_ESTRATTORE_BIT",
    "bitIndex": 0
}
},
"sceneTrue": "CappaOn",
"sceneFalse": "CappaOff"
}
]
}

```

Bagno Umido (TrendRule)

```

{
    "name": "BagnoUmidoTrend",
    "type": "trend",
    "intervalMs": 20000,
    "trend": {
        "area": "SENSORE_BAGNO_HUM",
        "scale": 10,
        "threshold": 65,
        "trend": "rising",
        "tofMinutes": 5
    },
    "sceneTrue": "BagnoUmido",
    "sceneFalse": "BagnoNormale"
}

```

Esempio 2 — Automazioni Security (Intrusione + Finestre Aperte)

(*BitmaskRule + MultiRule*)

Scene

```
{
  "scenes": [
    {
      "name": "AllarmeIntrusione",
      "actions": [
        { "area": "SIRENA", "value": 1 },
        { "area": "LUCI_ALLARME", "value": 1 }
      ]
    },
    {
      "name": "AllarmeStop",
      "actions": [
        { "area": "SIRENA", "value": 0 },
        { "area": "LUCI_ALLARME", "value": 0 }
      ]
    }
  ]
}
```

Regola bitmask (intrusione)

```
{
  "name": "Intrusione",
  "type": "bitmask",
  "intervalMs": 1000,
  "bitmask": {
    "area": "SECURITY_BITS",
    "bitIndex": 0
  },
  "sceneTrue": "AllarmeIntrusione",
  "sceneFalse": "AllarmeStop"
}
```

```
}
```

MultiRule (finestre aperte + porta aperta)

```
{
  "name": "PerimetroAperto",
  "type": "multi",
  "intervalMs": 2000,
  "multi": {
    "logic": "OR",
    "conditions": [
      { "area": "WINDOWS_OPEN", "op": ">", "threshold": 0 },
      { "area": "DOORS_OPEN", "op": ">", "threshold": 0 }
    ]
  },
  "sceneTrue": "AllarmeIntrusione",
  "sceneFalse": "AllarmeStop"
}
```

Esempio 3 — Automazioni Power (Load Shedding)

(*ThresholdRule + TimeRule*)

Scene

```
{
  "scenes": [
    {
      "name": "StaccaCarichi",
      "actions": [
        { "area": "LOAD1_CMD", "value": 0 },
        { "area": "LOAD2_CMD", "value": 0 }
      ]
    },
    {
      "name": "AttaccaCarichi",
      "actions": [
        { "area": "LOAD1_CMD", "value": 1 },
        { "area": "LOAD2_CMD", "value": 1 }
      ]
    }
  ]
}
```

Regola threshold (superamento limite)

```
{
  "name": "SuperamentoLimite",
  "type": "threshold",
  "intervalMs": 2000,
  "threshold": {
    "area": "GRID_POWER",
    "op": ">",
    "threshold": 3000
  },
  "sceneTrue": "StaccaCarichi",
}
```

```
"sceneFalse": "AttaccaCarichi"
}
```

Regola oraria (modalità notte)

```
{
  "name": "NightMode",
  "type": "time",
  "threshold": { "threshold": 22*60 },
  "sceneTrue": "StaccaCarichi"
}
```

Esempio 4 — Sequenza temporizzata (Luci scenografiche)

```
{
  "sequences": [
    {
      "intervalMs": 60000,
      "steps": [
        { "area": "LUCE1", "value": 1, "delayMs": 0 },
        { "area": "LUCE2", "value": 1, "delayMs": 2000 },
        { "area": "LUCE3", "value": 1, "delayMs": 2000 },
        { "area": "LUCE1", "value": 0, "delayMs": 5000 },
        { "area": "LUCE2", "value": 0, "delayMs": 2000 },
        { "area": "LUCE3", "value": 0, "delayMs": 2000 }
      ]
    }
  ]
}
```

Conclusione

Questo capitolo fornisce:

- esempi completi
- casi reali
- logiche composite
- trend, debounce, bitmask
- sequenze temporizzate

- automazioni HVAC, Security, Power, Weather
- un esempio full-system

Tutto senza alcun riferimento a sistemi esterni: **solo logica interna, buffer, scene e regole.**

Disclaimer Legale – Limitazione di Responsabilità

Il software **DomoManager**, la relativa documentazione tecnica e tutti i moduli associati (inclusi ma non limitati a: *DM.hpp*, *DMFrontend.hpp*, *DMSetup.hpp*, *DMDiagnostic.hpp*, automazioni, configurazioni e profili hardware) sono forniti **“così come sono”**, senza alcuna garanzia espressa o implicita.

L'autore e i collaboratori **non assumono alcuna responsabilità** per:

- danni diretti o indiretti
- malfunzionamenti
- perdita di dati
- interruzioni di servizio
- danni a cose, impianti o persone
- utilizzo improprio o non conforme del software
- modifiche non autorizzate al codice o alla configurazione
- integrazioni con dispositivi di terze parti non certificati

L'utente accetta che:

L'installazione, la configurazione e l'uso del software avvengono sotto la propria esclusiva responsabilità.

Il sistema è progettato per applicazioni domotiche e di automazione non critiche. Non deve essere utilizzato in contesti dove un malfunzionamento potrebbe causare:

- rischi per la sicurezza personale
- danni strutturali
- interruzioni di servizi essenziali
- condizioni pericolose per persone o animali

L'utente è tenuto a:

- verificare la correttezza delle configurazioni
- testare ogni automazione prima dell'uso reale
- assicurarsi che i dispositivi collegati siano conformi alle normative vigenti
- implementare adeguate misure di sicurezza elettrica e informatica

L'autore declina ogni responsabilità derivante da:

- modifiche al codice sorgente
- utilizzo in ambienti industriali o critici
- integrazioni con hardware non supportato
- errori di cablaggio o installazione
- uso improprio delle automazioni o delle scene

L'uso del software implica l'accettazione integrale del presente disclaimer.

DOMO Manager è ciò che davvero conta:

*la continuità, la sicurezza, la sostenibilità e la qualità dell'ambiente in cui viviamo ogni
giorno...*
