



[DomoManager Pratico: Configurazione di base]

Sistema
Domotico su
piattaforma
Arduino



AUTORE: ANDREA LANDO

REVISIONE 1.0.0

DATA ULTIMA REVISIONE: 09.05.2026

Sommario

Prefazione	6
Architettura Generale del Sistema	7
Visione d'insieme.....	7
Livelli principali	7
Principi architetturali	8
Diagramma architetturale aggiornato	9
Il Main	10
Ruolo del Main.....	10
Struttura del Setup	10
Loop principale	11
Filosofia progettuale del Main.....	12
Esempio completo di Main (versione aggiornata).....	13
Il FrontendEngine	14
Ruolo del FrontendEngine	14
Moduli orchestrati dal FrontendEngine.....	14
Setup del FrontendEngine	16
Loop del FrontendEngine.....	17
Diagramma del ciclo Frontend (aggiornato).....	18
Initialization (DMSetup.hpp) – Modello Dichiarativo	19
Struttura della configurazione dichiarativa	19
4.3 Come il Frontend costruisce il sistema	22
Pipeline di inizializzazione aggiornata	23
DomoManager (DM.hpp) – Cuore Logico del Sistema	24
Ruolo del DomoManager.....	24
Struttura interna del DomoManager.....	24
Ciclo Modbus TCP aggiornato.....	27
Gestione errori e retry	27
Watchdog e spike detection	28
FullCycleCallback e ActivityLoop.....	28
Diagramma interno del DomoManager	29
Buffer e Mappatura delle Aree.....	30
Ruolo del Buffer nel Sistema	30
Struttura interna di un'area del buffer	30
Come viene costruito il Buffer	31
Aree fisiche e aree virtuali	31
Flusso dati del Buffer	32

Mappatura Modbus → Buffer	33
Routing, Toggle e Split-Out	33
Diagnostica del Buffer	33
Esempio completo di definizione area	34
DeviceProfiles e DeviceManager	35
Ruolo dei DeviceProfiles	35
Profili custom	35
DeviceManager – Ruolo e Funzionamento	36
Registrazione dei dispositivi	36
Mappatura Modbus → Buffer	36
Gestione errori e retry	37
Calcolo massimo read size	37
Diagnostica dei dispositivi	38
Pipeline completa di gestione dispositivi	38
Automazioni (JSON)	39
Introduzione al modello di automazione	39
Struttura generale del JSON	39
Tipologie di regole supportate	41
8.4 Esempi reali dal tuo AUTOMATION_JSON	43
Best Practices per automazioni	44
Motori Applicativi – Guida Pratica all’Uso nel Sistema	45
HVACEngine – Uso pratico	45
WeatherEngine – Uso pratico	46
PowerEngine – Uso pratico	47
PowerSupervisorEngine – Uso pratico	47
SecuritySensorEngine – Uso pratico	48
AEEEngine – Uso pratico	49
MQTT Engine – Uso pratico	50
Diagnostica – Uso Pratico nel Sistema Reale	51
Watch Areas – Monitoraggio in tempo reale	51
ReportDeviceProfiles – Verifica profili hardware	51
ReportDeviceAreas – Verifica mappatura Modbus → Buffer	52
ReportMissingBufferDefinitions – Aree mancanti	53
Diagnostica dei motori applicativi	53
Logging non-bloccante	54
Watchdog – Diagnostica prestazioni	54
Diagnostica del ciclo Modbus	55

Diagrammi Architeturali (Aggiornati e Pratici)	56
Architettura Generale del Sistema	56
Pipeline Modbus TCP (Pratica)	58
Ciclo Frontend (Runtime Reale).....	59
Struttura Interna del DomoManager.....	60
Flusso Automazioni (Pratico)	61
Flusso Security (Pratico)	62
Flusso Power (Pratico)	63
Flusso Weather (Pratico)	64
Appendice Tecnica Completa	65
Strutture Dati Fondamentali.....	65
Tipi di Sensori.....	66
Tipi di Automazioni	66
Glossario Tecnico.....	66
Note di Implementazione	66
Setup dell'IDE Arduino.....	68
01	68
02	68
03	69
04	70
05	70
06	70
07	70
08	70
Disclaimer Legale – Limitazione di Responsabilità	72

Prefazione

L'evoluzione dei sistemi domotici moderni richiede un equilibrio sempre più raffinato tra affidabilità, modularità e capacità di adattarsi a scenari complessi. In questo contesto nasce **DomoManager**, un'architettura progettata per offrire un controllo completo e deterministico di dispositivi, sensori, automazioni e comunicazioni Modbus TCP, mantenendo al tempo stesso una struttura chiara, scalabile e facilmente estendibile. Questo manuale è dedicato a chi desidera comprendere le fondamenta del sistema, esplorando i tre livelli che costituiscono la colonna portante dell'intera piattaforma: **Main**, **Frontend** e **Setup**.

Il progetto è stato concepito con un obiettivo preciso: separare in modo netto la logica applicativa dalla logica di orchestrazione e dalla configurazione. Tale separazione consente di ottenere un sistema robusto, in cui ogni componente ha responsabilità ben definite e nessun modulo svolge compiti che non gli competono. Il risultato è un ecosistema che può crescere nel tempo senza perdere leggibilità, mantenibilità o stabilità operativa.

L'evoluzione dei sistemi domotici moderni richiede un equilibrio sempre più raffinato tra affidabilità, modularità e capacità di adattarsi a scenari complessi. In questo contesto nasce **DomoManager**, un framework professionale per la gestione di dispositivi Modbus TCP, sensori cablati, automazioni, logiche di controllo e motori applicativi avanzati (HVAC, Weather, Power, Security).

L'architettura è stata progettata con tre obiettivi fondamentali:

- **Separazione totale delle responsabilità**
- **Determinismo e non-bloccante**
- **Configurazione completamente dichiarativa**

Il sistema è organizzato in tre livelli principali:

1. **Main** – minimale, senza logica applicativa
2. **FrontendEngine** – orchestratore del runtime
3. **DomoManager** – cuore logico del sistema
4. **Setup (Initialization)** – definizione dichiarativa dell'intero impianto

Architettura Generale del Sistema

Visione d'insieme

DomoManager è un ecosistema modulare basato su librerie C++ che implementano:

- gestione dispositivi Modbus TCP
- buffer centralizzato
- automazioni dinamiche
- motori applicativi (HVAC, Weather, Power, Security)
- diagnostica avanzata
- scheduler asincrono
- watchdog con spike detection
- MQTT e AEE Bridge
- gestione sensori cablati

L'intero sistema è costruito tramite una configurazione dichiarativa contenuta in **DMSetup.hpp**, senza codice procedurale.

Livelli principali

Livello 1 – Main

- Punto di ingresso
- Nessuna logica applicativa
- Avvia il FrontendEngine
- Configura rete e parametri base

Livello 2 – FrontendEngine

- Orchestratore del runtime
- Gestisce Modbus, MQTT, Weather, HVAC, Power, Security
- Esegue il ciclo deterministico
- Coordina TaskEngine e diagnostica

Livello 3 – DomoManager

- Buffer
- DeviceManager
- AutomationEngine
- AsyncScheduler
- WiredSensorsManager
- Logic (toggle, split, routing, thresholds)
- Watchdog
- IpManager

- DeviceProfiles

Livello 4 – Setup (Initialization dichiarativa)

- DevicesConfig
- AreasConfig
- Routes
- Toggles
- Splits
- Automazioni JSON
- Configurazioni HVAC, Weather, Power, Security
- DiagnosticConfig
- Averages

Principi architetturali

Separazione delle responsabilità

Ogni modulo ha un ruolo preciso e non invade gli altri.

Determinismo

Ogni ciclo è misurabile, prevedibile e monitorato dal watchdog.

Non-bloccante

Nessuna funzione può bloccare il loop principale.

Scalabilità

Supporto per centinaia di dispositivi e migliaia di aree buffer.

Configurazione dichiarativa

Nessuna inizializzazione procedurale: tutto è definito in DMSetup.hpp.

Diagnostica integrata

Ogni componente espone report dettagliati.

Diagramma architetturale aggiornato



```

|           (DMSetup.hpp dichiarativo)           |
| - DevicesConfig                                |
| - AreasConfig                                  |
| - Routes / Toggles / Splits                    |
| - Automazioni JSON                             |
| - Configurazioni HVAC / Weather / Power / Security |
| - DiagnosticConfig                             |
| - Averages                                     |
+-----+

```

Il Main

Ruolo del Main

Il *Main* rappresenta il punto di ingresso dell'intero sistema. È progettato per essere **estremamente minimale**, privo di logiche applicative e completamente dedicato a:

- inizializzare l'hardware di base
- configurare la rete
- costruire la struttura FrontendConfig
- avviare il FrontendEngine
- delegare il ciclo di vita al livello superiore

Questa scelta architetturale garantisce:

- avvisi ripetibili e prevedibili
- assenza di logiche duplicate
- separazione totale tra configurazione e runtime
- debugging immediato

Il Main non contiene alcuna automazione, nessuna gestione sensori, nessuna logica di controllo. Tutto ciò appartiene ai livelli superiori.

Struttura del Setup

Il `setup()` del Main ha tre compiti fondamentali:

1. **Inizializzare la seriale e il logging**
2. **Costruire la configurazione completa del sistema** tramite FrontendConfig
3. **Avviare il FrontendEngine**

Esempio reale (semplificato)

```
void setup() {
    Serial.begin(19200);

    FrontendConfig cfg;

    // Configurazione rete
    cfg.net.mac = {0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED};
    cfg.net.ip = IPAddress(192, 168, 12, 202);
    cfg.net.gateway = IPAddress(192, 168, 12, 1);
    cfg.net.subnet = IPAddress(255, 255, 255, 0);

    // Configurazione Modbus, MQTT, HVAC, Weather, Power, Security...
    // (tutto definito in DMSetup.hpp)

    // Caricamento configurazione dichiarativa
    cfg.domoManager = domoConfig;

    // Avvio Frontend
    FrontendEngine::Setup(cfg);
}
```

Il Main non deve contenere:

- logiche di automazione
- gestione diretta del buffer
- gestione diretta dei dispositivi
- logiche di sicurezza
- logiche HVAC, Power, Weather

Tutto questo è responsabilità del FrontendEngine e dei motori applicativi.

Loop principale

Il loop() del Main è volutamente ridotto all'essenziale:

```
void loop() {
    FrontendEngine::Loop();
}
```

Cosa fa il loop del Main

- delega completamente al FrontendEngine

- non contiene alcuna logica
- non gestisce sensori
- non scrive nel buffer
- non esegue automazioni

Perché è così minimale

Per garantire:

- determinismo
- stabilità
- assenza di blocchi
- separazione totale tra livelli
- possibilità di sostituire il FrontendEngine senza toccare il Main

Filosofia progettuale del Main

Il Main è progettato per essere:

- **stateless**
- **privo di logica**
- **privo di dipendenze applicative**
- **semplice da leggere e mantenere**

Il suo unico compito è avviare il sistema.

Questa filosofia permette:

- portabilità del codice
- aggiornamenti del core senza modificare il Main
- debugging immediato
- avvio sempre coerente

Esempio completo di Main (versione aggiornata)

```
#include <Arduino.h>
#include "DMFrontend.hpp"
#include "DMSetup.hpp"

void setup() {
    Serial.begin(19200);

    FrontendConfig cfg;

    // --- Config rete ---
    cfg.net.mac = {0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED};
    cfg.net.ip = IPAddress(192, 168, 12, 202);
    cfg.net.gateway = IPAddress(192, 168, 12, 1);
    cfg.net.subnet = IPAddress(255, 255, 255, 0);

    // --- Configurazione dichiarativa ---
    cfg.domoManager = domoConfig;

    // --- Avvio Frontend ---
    FrontendEngine::Setup(cfg);
}

void loop() {
    FrontendEngine::Loop();
}
```

Questo è tutto ciò che serve. Il resto del sistema viene costruito automaticamente dal FrontendEngine.

Il FrontendEngine

Ruolo del FrontendEngine

Il **FrontendEngine** è il livello superiore dell'intero sistema. È il *direttore d'orchestra* che coordina:

- Modbus TCP (lettura/scrittura dispositivi)
- DomoManager (buffer, automazioni, scheduler)
- Motori applicativi (HVAC, Weather, Power, Security)
- MQTT Engine
- AEE Engine
- TaskEngine
- Diagnostica
- Watchdog

Il FrontendEngine **non contiene logica applicativa**: non decide cosa fare, ma *fa funzionare* chi deve decidere.

Il suo compito è garantire:

- un ciclo deterministico
- nessun blocco
- orchestrazione dei moduli
- gestione degli errori
- stabilità del runtime
- integrazione tra dispositivi, buffer e automazioni

Moduli orchestrati dal FrontendEngine

Il FrontendEngine coordina una serie di sottosistemi specializzati. Ogni modulo ha responsabilità precise e indipendenti.

DomoManager

Il cuore logico del sistema:

- buffer centralizzato
- DeviceManager
- AutomationEngine
- AsyncScheduler
- WiredSensorsManager
- Logic (toggle, split, routing, thresholds)
- Watchdog
- IpManager
- DeviceProfiles

HVACEngine

Gestisce:

- zone termiche
- pompa di calore
- logiche di comfort
- diagnostica HVAC

WeatherEngine

Gestisce:

- stazione meteo
- eventi (pioggia, vento, giorno/notte)
- allarmi meteo
- automazioni ambientali

PowerEngine

Gestisce:

- carichi elettrici
- limiti soft/hard
- auto-tuning
- forecast solare
- thermal load

PowerSupervisorEngine

Gestisce:

- tensione principale
- 12V / 24V
- fault elettrici
- battery mode
- callback allarmi

SecuritySensorEngine

Gestisce:

- sensori cablati
- zone
- tamper
- flood, smoke, intrusion
- bitmask di sicurezza
- comandi di engage/enable

AEEEngine

Gestisce:

- variabili AEE
- registrazione dinamica

- mapping verso MQTT

MQTT Engine

Gestisce:

- pubblicazione variabili
- integrazione Home Assistant
- mapping AEE → MQTT

TaskEngine

Round-robin deterministico per:

- scheduler
- automazioni
- watchdog
- diagnostica
- gestione errori

Setup del FrontendEngine

Il setup del FrontendEngine costruisce l'intero sistema.

Fasi del Setup

1. **Configurazione rete Ethernet**
2. **Avvio Modbus TCP server**
3. **Creazione DomoManager**
4. **Caricamento configurazione dichiarativa (DMSetup.hpp)**
5. **Setup motori applicativi**
6. **Setup MQTT / AEE**
7. **Setup diagnostica**
8. **Attivazione watchdog**

Esempio reale (semplificato)

```
FrontendEngine::Setup(cfg);
```

Il FrontendEngine:

- costruisce il DomoManager
- applica DevicesConfig
- applica AreasConfig
- applica Routes/Toggles/Splits
- parse AUTOMATION_JSON
- inizializza HVAC, Weather, Power, Security
- avvia MQTT e AEE
- attiva diagnostica e watchdog

Loop del FrontendEngine

Il loop del FrontendEngine è il *runtime reale* del sistema. È progettato per essere:

- deterministico
- non-bloccante
- misurabile
- stabile anche in caso di errori Modbus

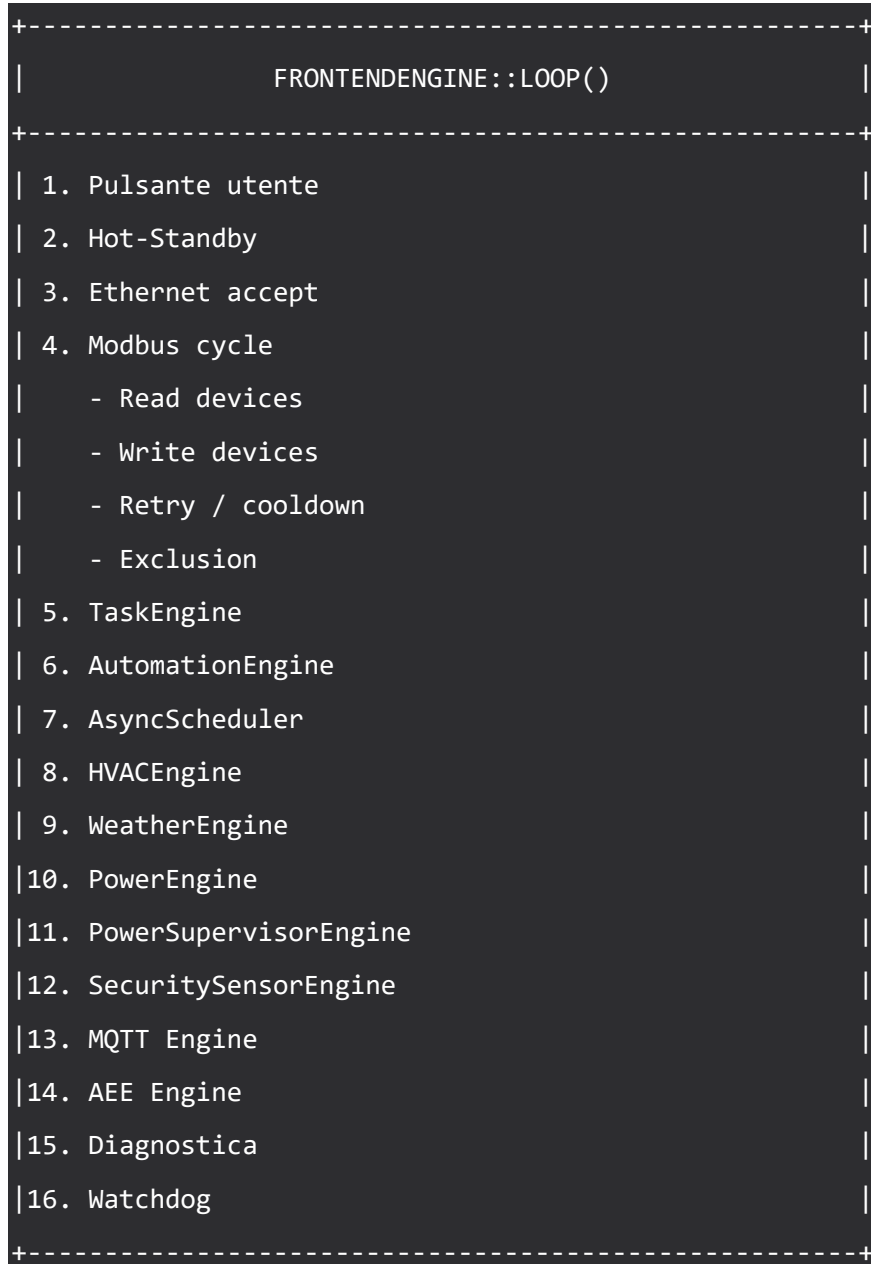
Sequenza operativa del loop

1. **Gestione pulsante utente**
2. **Gestione Hot-Standby (master/slave)**
3. **Accettazione client Ethernet**
4. **Ciclo Modbus TCP**
 - lettura dispositivi
 - scrittura dispositivi
 - gestione retry
 - esclusione dispositivi difettosi
5. **Esecuzione TaskEngine**
6. **Esecuzione AutomationEngine**
7. **Esecuzione AsyncScheduler**
8. **Aggiornamento motori applicativi**
 - HVACEngine
 - WeatherEngine
 - PowerEngine
 - PowerSupervisorEngine
 - SecuritySensorEngine
9. **MQTT Engine**
10. **AEE Engine**
11. **Diagnostica**
12. **Watchdog**

Esempio reale di loop

```
void loop() {  
    FrontendEngine::Loop();  
}
```

Diagramma del ciclo Frontend (aggiornato)



Initialization (DMSetup.hpp) – Modello Dichiarativo

Concetto chiave: inizializzazione completamente dichiarativa

Nelle versioni precedenti di DomoManager, l'inizializzazione avveniva tramite funzioni procedurali:

- mapDevices()
- InitBuffer()
- InitScenes()
- InitAutomations()
- InitScheduler()

Queste funzioni **non esistono più**.

L'intero sistema viene ora costruito tramite una **struttura costante**, definita in *DMSetup.hpp*:

```
static const DomoManagerConfig domoConfig = { ... };
```

Questa struttura descrive:

- dispositivi Modbus
- aree del buffer
- routing
- toggle
- split-out
- automazioni JSON
- configurazioni HVAC, Weather, Power, Security
- diagnostica
- medie

Il FrontendEngine legge questa configurazione e costruisce automaticamente l'intero sistema.

Struttura della configurazione dichiarativa

La configurazione è composta da blocchi indipendenti, ognuno con responsabilità precise.

DevicesConfig

Definisce tutti i dispositivi Modbus del sistema:

- nome
- IP
- Modbus ID
- profilo hardware
- lista aree buffer
- retry
- priorità

Esempio reale dal tuo codice:

```
static const DomoManagerConfig::Devices mainDevicesConfig = {
    {
        { "Lettore consumi - Quadro P1", WaveShareP1_Addr, 1, "LE_01MQ",
          { 10, 11, 12, 13, 14 }, 3, Low
        },
        ...
    }
};
```

Il DeviceManager costruisce automaticamente:

- GenericPrgDevice
- mapping Modbus → buffer
- diagnostica
- retry e cooldown
- esclusione dispositivi difettosi

AreasConfig

Definisce tutte le aree del buffer:

- ID area
- forwardArea
- label
- flag (readable, writable, virtual)
- soglie analogiche

Esempio reale:

```
{ AREA_PMETER_VOLTAGE, 0, "Lettura tensione", {true,false,false}, {50,200} },
```

Il BufferEngine genera automaticamente:

- aree fisiche
- aree virtuali
- diagnostica (duplicati, mancanti, non inizializzate)

RoutesConfig

Definisce logiche immediate buffer → buffer.

Esempio reale:

```
{
  "Bagno - Interruttore luce",
  BAGNO_IN_P1,
  {
    { 1, { { RELAY_LED_BAGNO, 1 }, ... } },
    { 0, { { RELAY_LED_BAGNO, 0 }, ... } }
  }
}
```

Il RouteEngine carica automaticamente tutte le route.

TogglesConfig

Definisce logiche ON/OFF su fronte.

Esempio reale:

```
{ 54, { BAGNO_IN_P2 } },
```

Il ToggleManager gestisce:

- edge detection
- forwarding
- debounce interno

SplitsConfig

Replica un valore su più aree per un tempo massimo.

Esempio reale:

```
{ 78, {114,115}, 30000 },
```

Automazioni JSON

Le automazioni non si scrivono più in C++.

Sono definite in un JSON:

```
static const char* AUTOMATION_JSON = R"json(
{
  "scenes": [...],
  "rules": [...],
  "sequences": []
}
)json";
```

Il FrontendEngine:

- parse JSON

- crea scene
- crea regole
- crea automazioni dinamiche
- crea composite rules
- crea trend rules
- crea debounce rules
- crea scheduled rules

Configurazioni HVAC / Weather / Power / Security

Ogni motore applicativo ha una sezione dedicata:

- FrontendConfig::HVAC
- FrontendConfig::Weather
- FrontendConfig::Power
- FrontendConfig::PowerSupervisor
- FrontendConfig::Security

Esempio reale (Power):

Codice

```
cfg.power.limitSoft = 3000;
cfg.power.limitHard = 3500;
cfg.power.autoTune = true;
cfg.power.tuneIntervalMs = 60000;
```

DiagnosticConfig

Permette di attivare:

- report DeviceProfiles
- report DeviceAreas
- report MissingBufferDefinitions
- report automazioni
- report scheduler

Averages

Definisce gruppi di sensori per calcolo medie automatiche.

4.3 Come il Frontend costruisce il sistema

Il flusso reale è:

1. Il Main passa FrontendConfig al FrontendEngine
2. Il FrontendEngine legge cfg.domoManager
3. Costruisce il DomoManager
4. Applica:
 - DevicesConfig

- AreasConfig
 - Routes
 - Toggles
 - Splits
5. Inizializza il buffer
 6. Parse AUTOMATION_JSON
 7. Costruisce automazioni e scene
 8. Inizializza motori applicativi
 9. Attiva diagnostica
 10. Avvia watchdog

Non esiste più codice manuale.

Pipeline di inizializzazione aggiornata

```
+-----+
|          FRONTENGINE::SETUP()          |
+-----+
| 1. Setup rete Ethernet                  |
| 2. Setup Modbus TCP server              |
| 3. Costruzione DomoManager              |
| 4. Caricamento configurazione dichiarativa |
|   - DevicesConfig                      |
|   - AreasConfig                       |
|   - Routes / Toggles / Splits          |
|   - Automazioni JSON                   |
| 5. Inizializzazione motori applicativi  |
|   - HVACEngine                        |
|   - WeatherEngine                     |
|   - PowerEngine                       |
|   - PowerSupervisorEngine              |
|   - SecuritySensorEngine               |
| 6. Setup MQTT / AEE                    |
| 7. Setup diagnostica                   |
| 8. Attivazione watchdog                 |
+-----+
```

DomoManager (DM.hpp) – Cuore Logico del Sistema

Ruolo del DomoManager

Il **DomoManager** è il nucleo centrale dell'intero framework. Tutto ciò che riguarda:

- buffer
- dispositivi
- automazioni
- scheduler
- sensori cablati
- routing/toggle/split
- diagnostica
- watchdog
- gestione IP
- profili hardware

viene gestito qui.

Il FrontendEngine *orchestra*, ma è il DomoManager che *esegue*.

Struttura interna del DomoManager

Il DomoManager è composto da numerosi sottosistemi, ognuno con responsabilità precise.

Buffer

Il buffer è la struttura dati centrale del sistema.

Contiene:

- valori letti dai dispositivi
- valori scritti dalle automazioni
- valori provenienti dal pannello
- timestamp
- flag (readable, writable, virtual)
- soglie analogiche
- diagnostica

Il buffer è completamente dinamico e viene costruito da:

- DevicesConfig
- AreasConfig
- InitBuffer() interno

DeviceManager

Gestisce:

- dispositivi Modbus TCP
- retry

- cooldown
- esclusione dispositivi difettosi
- diagnostica
- mapping Modbus → buffer

Ogni dispositivo è un `GenericPrgDevice`.

Logic

Contiene tre motori fondamentali:

- **ToggleManager**
- **SplitOutManager**
- **AnalogThresholdManager**

Questi motori implementano logiche immediate buffer → buffer.

AutomationEngine

Gestisce:

- regole
- scene
- sequenze
- automazioni dinamiche
- scheduled rules
- composite rules
- trend rules
- debounce rules

È alimentato dal JSON definito in `DMSetup.hpp`.

AsyncScheduler

Scheduler asincrono che gestisce:

- job temporizzati
- sequenze
- operazioni ritardate
- automazioni complesse

WiredSensorsManager

Gestisce:

- sensori cablati
- zone
- tamper
- flood, smoke, intrusion
- aggregazione stato
- bitmask di sicurezza

È utilizzato dal SecuritySensorEngine.

Watchdog

Sistema avanzato di monitoraggio prestazioni:

- EMA Q8.8
- spike detection
- overload detection
- logging intelligente
- misurazione tempi reali

Monitora:

- somethingChanged
- route
- activityLoop
- updateCycle

IpManager

Gestisce:

- lista IP dei dispositivi
- retry
- cooldown
- esclusione automatica
- rotazione round-robin

DeviceProfiles

Registro dei profili hardware:

- DI
- DO
- AI
- AO
- coil
- discrete
- holding
- input

I profili definiscono la struttura Modbus dei dispositivi.

CalcolatoreMedie

Gestisce:

- medie mobili
- gruppi di sensori
- smoothing dei valori

Ciclo Modbus TCP aggiornato

Il ciclo Modbus è eseguito dal DomoManager tramite Update().

Pipeline del ciclo Modbus

1. Selezione IP corrente
2. Verifica stato (OK, cooldown, excluded)
3. Lettura registri Modbus
4. Scrittura registri Modbus
5. Aggiornamento buffer
6. Routing / Toggle / Split
7. Automazioni
8. Scheduler
9. Diagnostica
10. Watchdog

Gestione errori

Il sistema gestisce automaticamente:

- retry
- cooldown
- esclusione IP
- logging errori
- ripristino automatico

Fine ciclo IP

Quando tutti gli IP sono stati interrogati:

- ipCycleCompleted = true
- viene eseguito il TaskEngine
- viene aggiornato il watchdog
- viene aggiornato il buffer AREA_SYSTEM_RUNNING_T

Gestione errori e retry

Il DomoManager implementa un sistema avanzato di gestione errori:

- ogni dispositivo ha un contatore errori
- dopo N errori → cooldown
- dopo ulteriori errori → exclusion
- dopo un periodo → retry automatico
- logging intelligente

Questo garantisce stabilità anche in presenza di:

- dispositivi difettosi

- rete instabile
- timeout Modbus

Watchdog e spike detection

Il watchdog misura:

- tempi di esecuzione
- medie mobili
- spike
- overload

Utilizza:

- EMA Q8.8
- soglie dinamiche
- logging intelligente

Esempio di log reale:

Codice

[SPIKE] updateCycle exec=42 avg=9 thr=32

FullCycleCallback e ActivityLoop

Il DomoManager esegue un ciclo completo solo quando:

- tutti gli IP sono stati interrogati
- il pannello Modbus è stato aggiornato

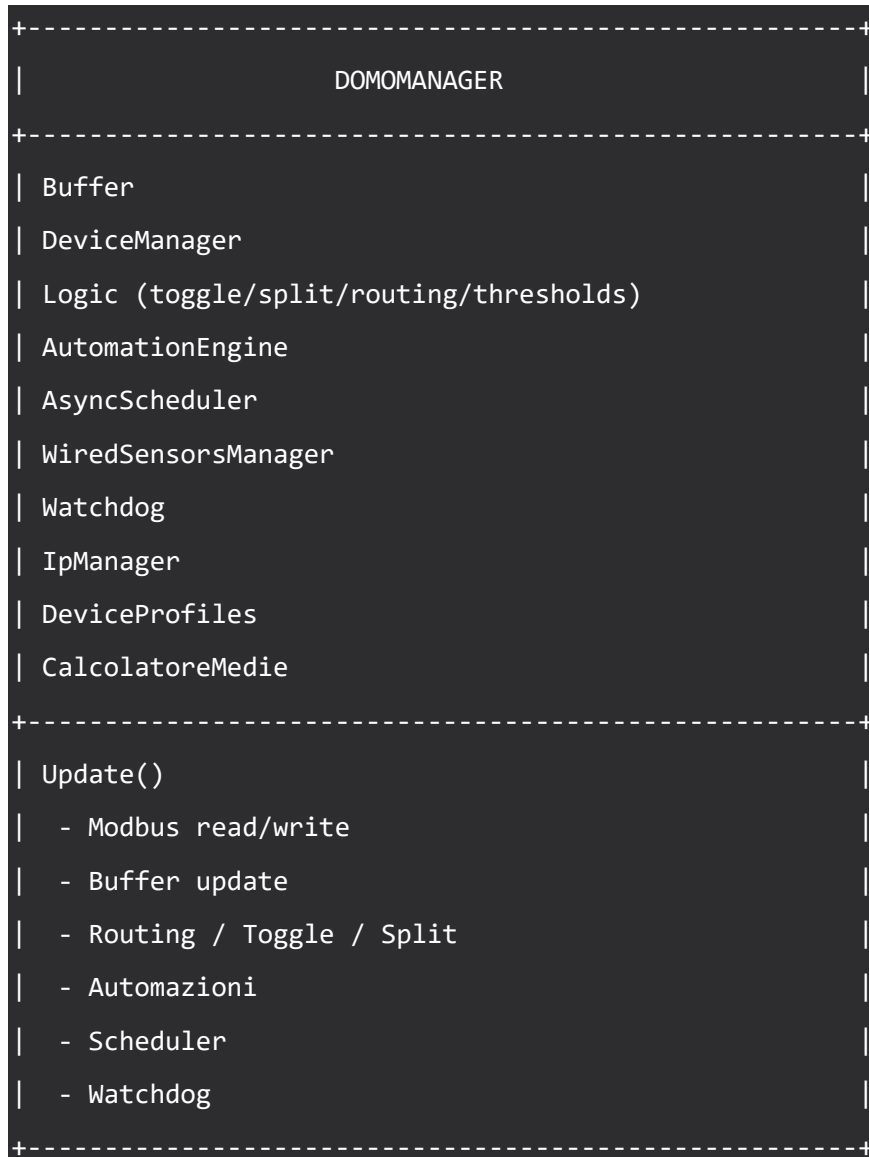
A quel punto:

- viene eseguito il TaskEngine
- vengono eseguite automazioni e scheduler
- vengono aggiornati i motori applicativi

Questo garantisce:

- determinismo
- sincronizzazione perfetta
- nessun conflitto tra Modbus e automazioni

Diagramma interno del DomoManager



Buffer e Mappatura delle Aree

Ruolo del Buffer nel Sistema

Il **Buffer** è la struttura dati centrale di DomoManager. Tutto ciò che entra o esce dal sistema passa attraverso il buffer:

- valori letti dai dispositivi Modbus
- valori scritti dalle automazioni
- valori provenienti dal pannello
- valori inviati ai motori applicativi
- valori pubblicati via MQTT
- valori utilizzati dal SecuritySensorEngine
- valori utilizzati dal PowerEngine
- valori utilizzati dal WeatherEngine

Il buffer è quindi il **punto di verità** dell'intero sistema.

Struttura interna di un'area del buffer

Ogni area del buffer è definita da una struttura AreaConfig:

- **area** → ID numerico
- **forwardArea** → area di destinazione per routing immediato
- **label** → descrizione umana
- **flags** → proprietà dell'area
- **threshold** → soglie analogiche

Flags disponibili

Flag	Significato
readable	il valore può essere letto dal pannello
writable	il valore può essere scritto dal pannello
virtualArea	l'area non è mappata a nessun dispositivo fisico

Threshold analogici

Ogni area può avere:

- soglia bassa
- soglia alta

Utilizzati da:

- AnalogThresholdManager
- automazioni
- diagnostica

Esempio reale:

Codice

```
{ AREA_PMETER_VOLTAGE, 0, "Lettura tensione", {true,false,false}, {50,200} },
```

Come viene costruito il Buffer

Il buffer viene costruito automaticamente da:

1. **DevicesConfig**
2. **AreasConfig**
3. **InitBuffer()** interno del **DomoManager**

Pipeline di costruzione

```
DevicesConfig → mappatura aree fisiche  
AreasConfig → definizione proprietà aree  
InitBuffer() → creazione buffer dinamico  
Auto-Virtual → marcatura aree non mappate  
Diagnostica → verifica duplicati e mancanti
```

Aree fisiche e aree virtuali

Aree fisiche

Sono aree mappate a:

- coil Modbus
- discrete input
- holding register
- input register

Provengono da:

- profili hardware
- DevicesConfig

Aree virtuali

Il sistema identifica automaticamente tutte le aree che:

- sono definite in AreasConfig
- **non** sono mappate da alcun dispositivo

E le marca come **virtualArea = true**.

Esempio reale dal tuo log:

```
InitBuffer: Aree virtuali auto-determinate: 222 223 225 226 ...
```

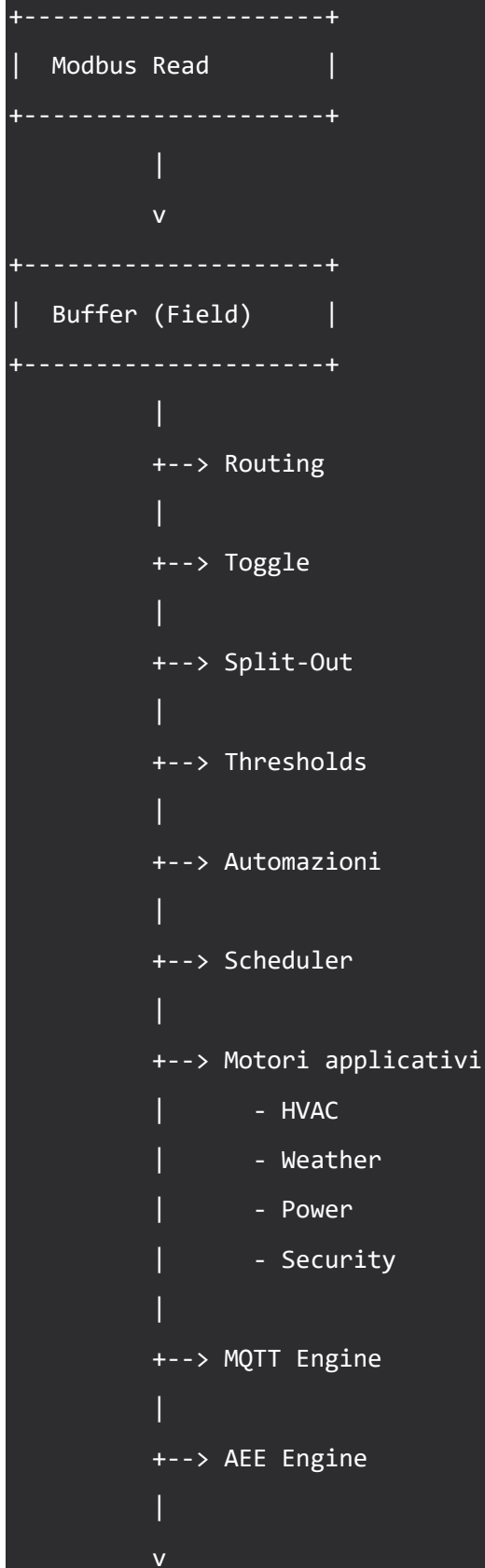
Le aree virtuali sono fondamentali per:

- automazioni
- comandi logici
- bitmask
- motori applicativi
- MQTT

Flusso dati del Buffer

Il buffer è al centro di un flusso dati complesso ma deterministico.

Pipeline completa



```
+-----+
|  Modbus Write  |
+-----+
```

Mappatura Modbus → Buffer

La mappatura avviene tramite:

- profilo hardware (es. MA01_XACX0440)
- lista aree in DevicesConfig

Esempio reale:

```
{ "Scheda 4DI+4DO - Scala", WaveShareP1_Addr, 2, "MA01_XACX0440",
  { AREA_INGRESSO_DOOR_ALM, 54, 55, 56, 57, 58, AREA_PLAFONIERA_EXT, 60 },
  3, High
}
```

Il profilo definisce:

- 4 DI
- 4 DO

La lista aree deve contenere **esattamente 8 aree**.

Il DeviceManager:

- associa ogni area al registro Modbus corretto
- gestisce retry e cooldown
- aggiorna il buffer

Routing, Toggle e Split-Out

Questi tre motori logici operano direttamente sul buffer.

Routing

Logica immediata:

```
se area X = 1 → scrivi Y=1, Z=0, ...
```

Toggle

Edge detection:

```
fronte di salita → inverti stato
```

Split-Out

Replica un valore su più aree per un tempo massimo.

Diagnostica del Buffer

Il DomoManager esegue automaticamente:

- rilevamento aree duplicate
- rilevamento aree non inizializzate
- rilevamento aree non mappate

- rilevamento aree virtuali
- logging soglie analogiche
- logging routing/toggle/split

Esempio reale:

```
ERRORE: Area inizializzata più volte: 75
```

Esempio completo di definizione area

```
{ AREA_CUCINA_PIR_ALM, 0, "PIR Cucina Allarme, NEGATO",
  {true,false,true}, {-1,-1} }
```

Significato:

- **AREA_CUCINA_PIR_ALM** → ID area
- **forwardArea = 0** → nessun routing diretto
- **label** → descrizione
- **flags** → readable, not writable, virtual
- **threshold** → nessuna soglia

DeviceProfiles e DeviceManager

Ruolo dei DeviceProfiles

I **DeviceProfiles** rappresentano la descrizione formale della struttura hardware dei dispositivi Modbus TCP utilizzati nel sistema. Ogni profilo definisce:

- il numero di canali
- il tipo di canale (DI, DO, AI, AO)
- il tipo di registro Modbus (Coil, Discrete, Holding, Input)
- l'indirizzo iniziale
- il numero di item
- il raggruppamento delle letture (ItemsPerCall)

I profili sono fondamentali perché permettono al sistema di:

- interpretare correttamente i registri Modbus
- mappare i valori nel buffer
- gestire retry e diagnostica
- garantire compatibilità tra dispositivi diversi

Esempio di profilo hardware

Un profilo reale (es. MA01_XACX0440) è definito così:

```
add("MA01_XACX0440", {  
    { DI, Discrete, 0, 4, 1 },  
    { DO, Coil,      0, 4, 1 }  
});
```

Significato:

- 4 ingressi digitali (Discrete Input)
- 4 uscite digitali (Coil)
- indirizzo iniziale 0
- lettura/scrittura 1 registro per chiamata

Profili custom

È possibile definire profili custom direttamente in *DMSetup.hpp*. Questa è la pratica consigliata perché:

- evita modifiche ai file core
- garantisce compatibilità con aggiornamenti futuri
- permette di aggiungere dispositivi non standard

Esempio:

```
profiles.add("MY_6DI_2DO", {  
    { DI, Discrete, 0, 6, 1 },  
    { DO, Coil,      0, 2, 1 }  
});
```

```
});
```

DeviceManager – Ruolo e Funzionamento

Il **DeviceManager** è responsabile della gestione di tutti i dispositivi Modbus TCP presenti nel sistema.

Compiti principali

- registrazione dispositivi
- gestione retry
- gestione cooldown
- esclusione dispositivi difettosi
- diagnostica
- mapping Modbus → buffer
- calcolo massimo read size
- gestione priorità

Ogni dispositivo è rappresentato da un oggetto `GenericPrgDevice`.

Registrazione dei dispositivi

La registrazione avviene tramite la sezione **DevicesConfig** in *DMSetup.hpp*.

Esempio reale:

```
{ "Scheda 4DI+4DO - Scala", WaveShareP1_Addr, 2, "MA01_XACX0440",  
  { AREA_INGRESSO_DOOR_ALM, 54, 55, 56, 57, 58, AREA_PLAFONIERA_EXT, 60 },  
  3, High  
}
```

Significato:

- nome dispositivo
- IP
- Modbus ID
- profilo hardware
- lista aree buffer
- retry
- priorità

Regola fondamentale

Il numero di aree deve essere **esattamente uguale** al numero totale di item del profilo.

Esempio:

Profilo MA01_XACX0440 → 4 DI + 4 DO = 8 item Lista aree → deve contenere 8 aree.

Mappatura Modbus → Buffer

La mappatura avviene automaticamente:

1. Il profilo definisce la struttura Modbus

2. DevicesConfig fornisce la lista aree
3. Il DeviceManager associa ogni registro Modbus all'area corrispondente

Esempio di mappatura

Profilo:

- DI[0] → Discrete Input 0
- DI[1] → Discrete Input 1
- DO[0] → Coil 0
- DO[1] → Coil 1

Lista aree:

```
{ 54, 55, 56, 57, 58, 59, 60, 61 }
```

Mappatura:

- DI[0] → area 54
- DI[1] → area 55
- DI[2] → area 56
- DI[3] → area 57
- DO[0] → area 58
- DO[1] → area 59
- DO[2] → area 60
- DO[3] → area 61

Gestione errori e retry

Il DeviceManager implementa un sistema avanzato di gestione errori:

- ogni dispositivo ha un contatore errori
- dopo N errori → cooldown
- dopo ulteriori errori → exclusion
- dopo un periodo → retry automatico
- logging intelligente

Esempio reale:

```
DeviceHasErrors: 2 devices in error
```

Calcolo massimo read size

Il DeviceManager calcola automaticamente la dimensione massima del buffer Modbus necessario per leggere tutti i dispositivi.

Questo valore viene utilizzato per:

- allocare mbReadBuffer
- ottimizzare le letture Modbus
- evitare overflow

Diagnostica dei dispositivi

Il sistema fornisce diagnostica avanzata:

ReportDeviceProfiles

Mostra:

- nome profilo
- numero canali
- numero item
- stato

ReportDeviceAreas

Verifica:

- coerenza mappatura
- duplicati
- aree mancanti

ReportMissingBufferDefinitions

Individua:

- aree mappate ma non definite in AreasConfig

Pipeline completa di gestione dispositivi

```
+-----+
|          DEVICEMANAGER          |
+-----+
| 1. Caricamento profili hardware |
| 2. Registrazione dispositivi      |
| 3. Mappatura Modbus → Buffer      |
| 4. Gestione retry                |
| 5. Gestione cooldown             |
| 6. Esclusione dispositivi difettosi |
| 7. Lettura Modbus                |
| 8. Scrittura Modbus              |
| 9. Diagnostica                   |
+-----+
```

Automazioni (JSON)

Introduzione al modello di automazione

Il sistema di automazioni di DomoManager è stato completamente riprogettato per essere:

- **dichiarativo**
- **scalabile**
- **leggibile**
- **compatibile con Home Assistant**
- **indipendente dal codice C++**
- **estendibile senza ricompilare**

Le automazioni non si scrivono più in C++. Sono definite in un unico blocco JSON:

```
static const char* AUTOMATION_JSON = R"json({
  "scenes": [...],
  "rules": [...],
  "sequences": []
})json";
```

Il FrontendEngine:

- esegue il parsing
- valida la struttura
- costruisce scene, regole e sequenze
- registra automazioni dinamiche
- collega tutto al buffer

Struttura generale del JSON

Il JSON è composto da tre sezioni principali:

1. **scenes**
2. **rules**
3. **sequences**

Scenes

Una *scene* è un insieme di azioni atomiche:

```
{
  "name": "CappaOn",
  "actions": [
    { "area": "AREA_ESTRATTORE_CUCINA", "value": 1 }
  ]
}
```

Le scene sono:

- riutilizzabili
- richiamabili da regole
- eseguite in modo atomico

Rules

Una *rule* è una logica condizionale che può essere:

- simple
- trend
- debounce
- composite
- bitmask
- scheduled
- dynamic

Esempio reale (trend):

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

Sequences

Una *sequence* è una serie di azioni temporizzate.

Esempio:

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

Tipologie di regole supportate

Simple Rule

Valuta direttamente un'area:

```
{ "type": "simple", "area": "AREA_CUCINA_PIR_ALM" }
```

Trend Rule

Analizza l'andamento di un valore nel tempo.

Usata per:

- umidità bagno
- umidità cucina
- temperatura

Esempio reale:

```
"trend": {
  "area": "SENSORE_BAGNO_HUM",
  "scale": 10,
  "threshold": 65,
  "trend": "rising",
  "tofMinutes": 5
}
```

Debounce Rule

Stabilizza segnali rumorosi.

Esempio reale:

```
{
  "type": "debounce",
  "name": "corrente",
  "area": "SENSORE_CUCINA_CORRENTE",
  "threshold": 1,
  "debounceMs": 10000,
  "tofMinutes": 10
}
```

Composite Rule

Combina più regole con logica:

- AND
- OR

Esempio reale (Cappa cucina):

```
"composite": {
  "logic": "OR",
  "inputs": [
    { "type": "debounce", ... },
    { "type": "trend", ... },
    { "type": "bitmask", ... }
  ],
  "output": {
    "area": "AREA_CUCINA_ESTRATTORE_BIT",
    "bitIndex": 0
  }
}
```

Bitmask Rule

Legge un singolo bit da un'area.

Esempio:

```
{
  "type": "bitmask",
  "area": "AREA_CUCINA_ESTRATTORE_BIT",
  "bitIndex": 2
}
```

Scheduled Rule

Regole basate sul tempo:

- ogni X minuti
- ogni ora
- ogni giorno

Esempio:

```
{
  "type": "scheduled",
  "cron": "0 */1 * * *",
  "sceneTrue": "LightsOff"
}
```

Dynamic Automation

Automazioni eseguite periodicamente.

Esempio reale:

```
"intervalMs": 20000
```

8.4 Esempi reali dal tuo AUTOMATION_JSON

Bagno Umido

Logica:

- se umidità sale oltre 65%
- per almeno 5 minuti
- attiva estrattore

Scene:

- BagnoUmido
- BagnoNormale

Cappa Cucina

Logica OR:

- corrente > soglia
- umidità in aumento
- comando manuale bitmask

Scene:

- CappaOn
- CappaOff

AutoOff

Logica AND:

- tutti i PIR attivi
- tutte le finestre chiuse

- porta ingresso chiusa

Scene:

- LightsOff

Best Practices per automazioni

- usare nomi chiari
- evitare logiche troppo complesse in una sola regola
- preferire composite rules
- usare trend per segnali ambientali
- usare debounce per segnali rumorosi
- usare bitmask per comandi manuali
- usare scene per raggruppare azioni

Motori Applicativi – Guida Pratica all’Uso nel Sistema

Questa sezione spiega **come funzionano davvero** i motori applicativi nel tuo runtime, **cosa fanno nel loop**, **quali dati leggono**, **quali aree buffer usano**, **quali callback generano** e **come diagnosticarli**.

È la parte che serve a chi deve:

- integrare nuovi sensori
- capire perché una logica si attiva
- verificare comportamenti anomali
- aggiungere nuove zone HVAC
- aggiungere nuovi segnali meteo
- aggiungere nuovi carichi elettrici
- capire come i motori interagiscono con il buffer

HVACEngine – Uso pratico

Cosa fa nel runtime

Nel loop del Frontend, HVACEngine:

1. legge le temperature delle zone
2. legge temperatura interna/esterna
3. legge stato finestra
4. legge orario HVAC (HeatPumpController::HVACTime)
5. aggiorna la pompa di calore
6. aggiorna le zone (setpoint, modalità, valvole)

Come viene chiamato

Dal Frontend:

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

Dove scrive

Il motore HVAC **non scrive direttamente nel buffer**. Scrive solo nei suoi oggetti interni (DMHVAC), che poi:

- attivano relè
- comandano valvole
- generano diagnostica

Come diagnosticarlo

Puoi ottenere un report completo:

```
HVACEngine::DiagnosticReport();
```

Mostra:

- stato compressore
- stato valvole
- modalità attuale

- temperatura target
- temperatura reale

Quando usarlo

- gestione PDC
- gestione zone termiche
- logiche di comfort
- integrazione con automazioni (es. spegnere HVAC se finestra aperta)

WeatherEngine – Uso pratico

Cosa fa nel runtime

Ogni ciclo:

1. legge i sensori meteo (pioggia, vento, temperatura, luminosità)
2. aggiorna lo stato interno
3. genera eventi (RainStart, WindGustStart, DayStart...)
4. genera allarmi (TempLow, WindHigh...)
5. attiva automazioni (es. chiusura finestre)

Callback pratici

Gli eventi sono gestiti così:

```
static void weatherEventHandler(WeatherEvent event)
```

Esempi reali:

- RainStart → chiudi finestre
- WindGustStart → chiudi finestre
- NightStart → chiudi finestre

Gli allarmi:

```
static void weatherAlarmHandler(const WeatherAlarm* alarms, int count)
```

Come diagnosticarlo

Log automatici:

```
[METEO] ☁ Inizia a piovere
[METEO] 🌀 Raffica di vento!
```

Quando usarlo

- automazioni ambientali
- protezione finestre
- logiche giorno/notte
- integrazione con sicurezza

PowerEngine – Uso pratico

Cosa fa nel runtime

Ogni ciclo:

1. legge potenza di rete (gridPower)
2. legge luminosità, temperatura esterna
3. calcola forecast solare
4. aggiorna auto-tuning
5. aggiorna carichi elettrici
6. aggiorna thermal load

Callback pratici

Il PowerManager genera eventi:

```
onLoadChange(name, state)
onLimitWarning(netPower, limit)
onLimitExceeded(netPower, limit)
onPowerError(code, msg)
onSuggestion(suggestion, severity, reason)
```

Esempi reali:

```
[POWER] [WARNING] net=3200 limit=3000
[POWER] [EXCEEDED] net=3800 limit=3500
```

Come diagnosticarlo

Puoi leggere lo stato:

```
PowerEngine::Get().dumpState();
```

Quando usarlo

- gestione carichi prioritari
- protezione da sovraccarico
- gestione estrattori, boiler, pompe
- logiche di risparmio energetico

PowerSupervisorEngine – Uso pratico

Cosa fa nel runtime

Ogni ciclo:

1. legge tensione principale
2. legge stato 12V / 24V
3. legge fault
4. legge battery mode
5. genera allarmi

Callback pratici

```
static void onAlarm(const String& name, long value)
```

Esempi reali:

```
[ALARM] MAIN_POWER = 0 → Mancanza tensione  
[ALARM] FAULT = 1
```

Quando usarlo

- blackout
- fault elettrici
- monitoraggio alimentatori
- gestione UPS

SecuritySensorEngine – Uso pratico

Cosa fa nel runtime

Ogni ciclo:

1. legge tutti i sensori cablati
2. resetta sensori in errore
3. rileva allarmi per tipo (RT, H24, MASK, LEN)
4. aggiorna zone (Cucina, Camera, Bagno, Ingresso)
5. aggiorna SystemManager (bitmask sicurezza)
6. genera log diagnostici

Callback pratici

Allarmi globali

```
onAnyAlarm(zone, type, sensors)
```

Allarmi per zona

```
onZoneAlarm("Cucina", ...)
```

Allarmi per tipo

```
onTypeAlarm(SensorChannelType::RT, ...)
```

Allarmi zona+tipo

```
onZoneTypeAlarm("Camera", RT, ...)
```

Bitmask sicurezza

SystemManager produce un bitmask:

Bit Significato

- 0 Intrusione
- 1 Intrusione H24
- 2 Allagamento

Bit Significato

- 3 Fumo
- 4 Finestre aperte
- 5 Porte aperte
- 6 Tamper

Come diagnosticarlo

Report completo:

```
SecuritySensorEngine::getSystem().DiagnosticReport();
```

Esempio reale:

```
Intrusione : FALSE  
Fumo       : TRUE  
Bitmask    : 8
```

Quando usarlo

- antifurto
- antiallagamento
- antincendio
- logiche di sicurezza avanzate
- integrazione con automazioni

AEEEngine – Uso pratico

Cosa fa

- registra variabili AEE
- le espone al MQTT Engine
- permette mapping dinamico

Come si usa

Nel setup:

```
AEEEngine::instance().Setup(cfg.bridge.aee);
```

Per accedere:

```
AEEEngine::AEE().forEach(...)
```

Quando usarlo

- variabili custom
- integrazione con Home Assistant
- telemetria avanzata

MQTT Engine – Uso pratico

Cosa fa

- pubblica variabili buffer
- pubblica variabili AEE
- integra Home Assistant
- gestisce riconessioni

Come si configura

In DMSetup.hpp:

```
cfg.mqtt.vars = myVars;  
cfg.mqtt.varCount = N;
```

Ogni variabile:

```
{ "temp_cucina", "Temperatura Cucina", "°C", SENSOR, AREA_CUCINA_TEMP, 0.1f }
```

Quando usarlo

- dashboard
- telemetria
- integrazione HA
- logging esterno

Diagnostica – Uso Pratico nel Sistema Reale

La diagnostica è uno dei punti di forza del tuo DomoManager. Non è un modulo “accessorio”: è un **sistema di controllo industriale** che permette di:

- capire cosa sta succedendo
- individuare errori hardware
- verificare la coerenza della configurazione
- monitorare prestazioni e spike
- controllare lo stato dei dispositivi
- validare automazioni e routing
- analizzare sensori cablati e zone

Questa sezione spiega **come usarla davvero**, nel runtime.

Watch Areas – Monitoraggio in tempo reale

Le *Watch Areas* permettono di monitorare in tempo reale specifiche aree del buffer.

Come si configurano

In DMSetup.hpp:

```
diag.addArea(AREA_PMETER_POWER);  
diag.addArea(AREA_CUCINA_PIR_ALM);
```

Cosa succede nel runtime

Ogni volta che una di queste aree cambia valore:

- viene loggato il cambiamento
- viene mostrato il valore precedente e quello nuovo
- viene mostrata la sorgente (ToPanel, FromPanel, Field)

Esempio reale di log

```
[DIAG] Area 12 changed: 450 → 520 (Field)
```

Quando usarle

- debug sensori
- debug automazioni
- debug routing
- validazione configurazione

ReportDeviceProfiles – Verifica profili hardware

Serve per verificare che i profili hardware siano:

- corretti
- coerenti
- completi

Come si usa

Nel setup diagnostico:

```
if (cfg.diagnostic.reportDeviceProfiles)
    manager.getProfiles().Report();
```

Cosa mostra

- nome profilo
- numero canali
- numero item
- tipo registri
- coerenza interna

Quando usarlo

- aggiunta nuovo dispositivo
- debug mappatura aree
- verifica profili custom

ReportDeviceAreas – Verifica mappatura Modbus → Buffer

Questo report è fondamentale: ti dice se la mappatura tra profilo hardware e aree buffer è corretta.

Come si usa

Nel setup diagnostico:

```
if (cfg.diagnostic.reportDeviceAreas)
    manager.devices().ReportDeviceAreas();
```

Cosa mostra

Per ogni dispositivo:

- DI[0] → area X
- DI[1] → area Y
- DO[0] → area Z
- ecc.

Esempio reale

```
===== DEVICE AREAS =====
DI[0] → 54
DI[1] → 55
DO[0] → 58
DO[1] → 60
```

Quando usarlo

- aggiunta nuovo dispositivo
- debug di un sensore che non cambia
- debug di un relè che non si attiva

ReportMissingBufferDefinitions – Aree mancanti

Questo report individua:

- aree mappate da dispositivi
- ma non definite in AreasConfig

Come si usa

Nel setup diagnostico:

```
if (cfg.diagnostic.reportMissingBufferDefinitions)
    manager.getBuffer().ReportMissingDefinitions();
```

Esempio reale

```
ERRORE: Area 75 mappata da dispositivo ma non definita nel buffer
```

Quando usarlo

- dopo modifiche a DevicesConfig
- dopo aggiunta nuovi profili
- dopo refactor delle aree

Diagnostica dei motori applicativi

Ogni motore applicativo ha un proprio sistema di diagnostica.

HVACEngine

```
HVACEngine::DiagnosticReport();
```

Mostra:

- stato compressore
- stato valvole
- temperatura target
- temperatura reale

WeatherEngine

Log automatici:

```
[METEO] ☁ Inizia a piovere
[METEO] 🌪 Raffica di vento!
```

PowerEngine

Log automatici:

```
[POWER] [WARNING] net=3200 limit=3000
[POWER] [EXCEEDED] net=3800 limit=3500
```

PowerSupervisorEngine

```
[ALARM] MAIN_POWER = 0 → Mancanza tensione
```

SecuritySensorEngine

Report completo:

```
SecuritySensorEngine::getSystem().DiagnosticReport();
```

Mostra:

- intrusion
- flood
- smoke
- tamper
- bitmask

Logging non-bloccante

Il logger è progettato per:

- non bloccare mai il loop
- non rallentare il Modbus
- non interferire con automazioni

Livelli disponibili

- DEBUG
- INFO
- WARNING
- ERROR

Esempio

```
LOG_IF("POWER", "PowerManager configurato dinamicamente");
```

Watchdog – Diagnostica prestazioni

Il watchdog misura:

- tempi di esecuzione
- medie mobili
- spike
- overload

Esempio reale

```
[SPIKE] updateCycle exec=42 avg=9 thr=32
```

Quando usarlo

- debug rallentamenti
- debug congestione Modbus
- debug automazioni lente

Diagnostica del ciclo Modbus

Il ciclo Modbus è completamente tracciato.

Esempi reali

```
[MODBUS] Read OK from IP 192.168.12.203  
[MODBUS] Device 3 in cooldown  
[MODBUS] Excluding IP 192.168.12.205
```

Quando usarla

- dispositivi che non rispondono
- timeout
- errori di rete

Diagrammi Architeturali (Aggiornati e Pratici)

Questa sezione contiene **solo diagrammi**, completamente aggiornati al software reale. Sono pensati per essere incollati in Word senza perdere formattazione.

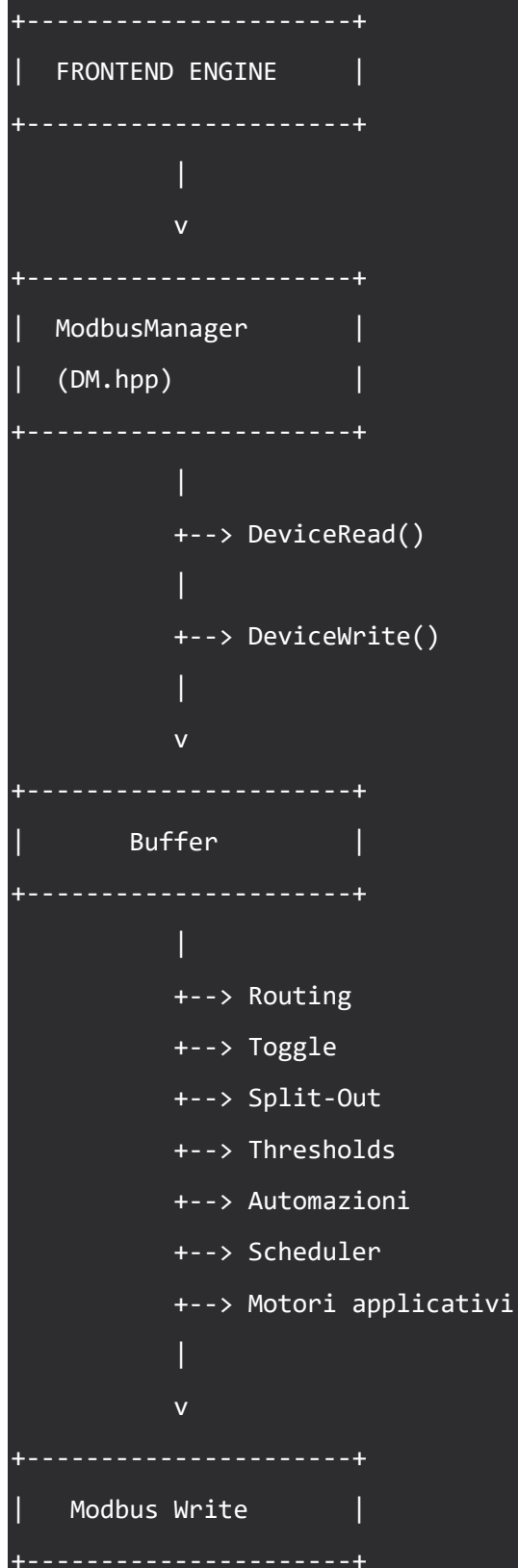
Architettura Generale del Sistema



v

```
+-----+
|           INITIALIZATION           |
|      (DMSetup.hpp dichiarativo)    |
| - DevicesConfig                    |
| - AreasConfig                      |
| - Routes / Toggles / Splits        |
| - Automazioni JSON                 |
| - Configurazioni HVAC / Weather /  |
|   Power / Security                 |
| - DiagnosticConfig                 |
| - Averages                         |
+-----+
```

Pipeline Modbus TCP (Pratica)



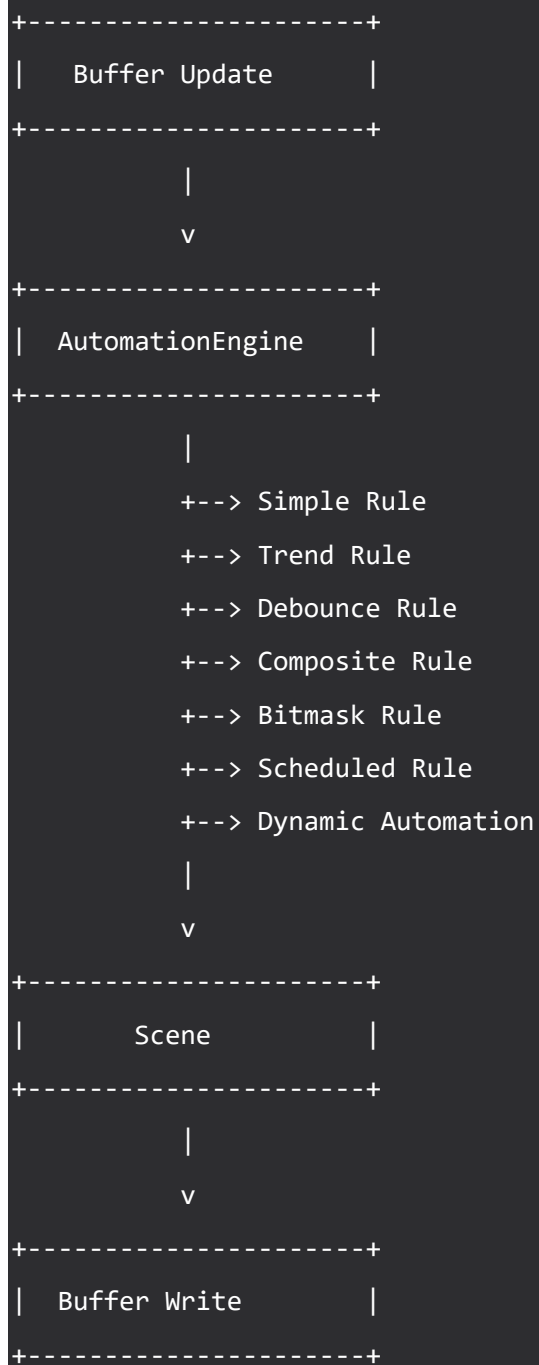
Ciclo Frontend (Runtime Reale)

```
+-----+
|          FRONTENDENGINE::LOOP()          |
+-----+
| 1. Pulsante utente                        |
| 2. Hot-Standby                           |
| 3. Ethernet accept                       |
| 4. Modbus cycle                          |
|   - Read devices                        |
|   - Write devices                       |
|   - Retry / cooldown                    |
|   - Exclusion                            |
| 5. TaskEngine                            |
| 6. AutomationEngine                      |
| 7. AsyncScheduler                       |
| 8. HVACEngine                           |
| 9. WeatherEngine                        |
|10. PowerEngine                           |
|11. PowerSupervisorEngine                 |
|12. SecuritySensorEngine                 |
|13. MQTT Engine                          |
|14. AEE Engine                           |
|15. Diagnostica                           |
|16. Watchdog                             |
+-----+
```

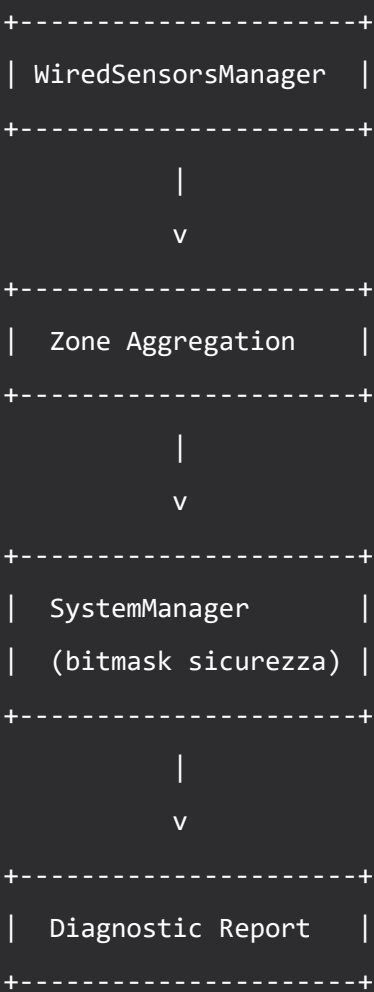
Struttura Interna del DomoManager

```
+-----+
|          DOMOMANAGER          |
+-----+
| Buffer                         |
| DeviceManager                 |
| Logic (toggle/split/routing/thresholds) |
| AutomationEngine              |
| AsyncScheduler                |
| WiredSensorsManager           |
| Watchdog                      |
| IpManager                     |
| DeviceProfiles                |
| CalcolatoreMedie              |
+-----+
| Update()                      |
| - Modbus read/write           |
| - Buffer update                |
| - Routing / Toggle / Split    |
| - Automazioni                 |
| - Scheduler                   |
| - Watchdog                    |
+-----+
```

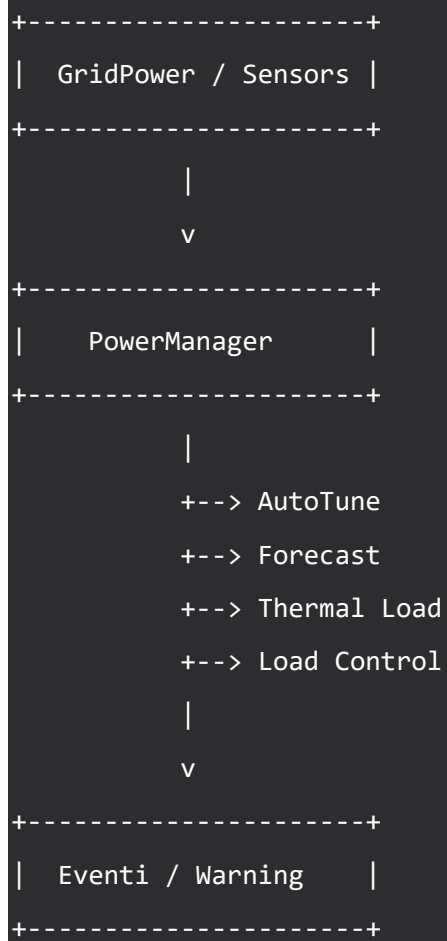
Flusso Automazioni (Pratico)



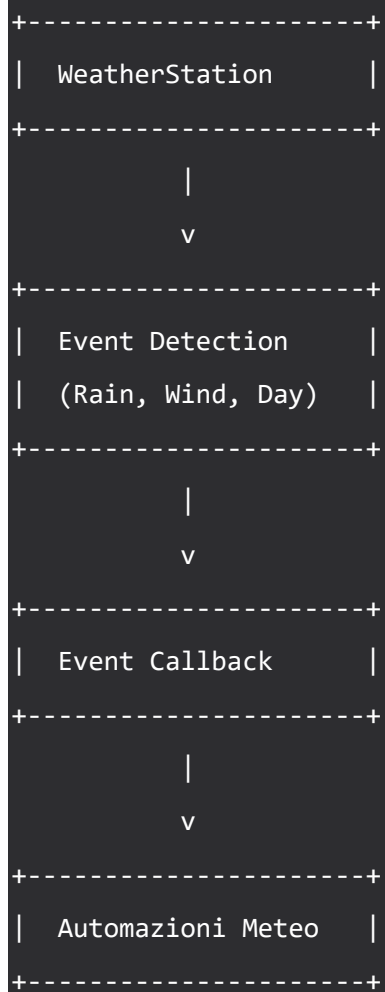
Flusso Security (Pratico)



Flusso Power (Pratico)



Flusso Weather (Pratico)



Appendice Tecnica Completa

Questa sezione contiene **solo ciò che serve davvero** quando si sviluppa o si estende il sistema.

Strutture Dati Fondamentali

GenericPrgDeviceChannel

```
struct GenericPrgDeviceChannel {  
    GenericPrgDeviceEnum type;    // DI, DO, AI, AO  
    GenericPrgDeviceHwEnum hwType; // Coil, Discrete, Hold, Input  
    int startingAddr;  
    int items;  
    int ItemsPerCall;  
};
```

Uso pratico:

- definisce come leggere/scrivere un blocco Modbus
- usato dai profili hardware

ProfileEntry

```
struct ProfileEntry {  
    const char* name;  
    std::vector<Channel> channels;  
};
```

Uso pratico:

- definisce la struttura di un dispositivo
- usato dal DeviceManager

Buffer::AreaConfig

```
struct AreaConfig {  
    int area;  
    int forwardArea;  
    String label;  
    AreaFlags flags;  
    ThresholdConfig threshold;  
};
```

Uso pratico:

- definisce proprietà dell'area
- usato da routing/toggle/split

Tipi di Sensori

Tipo	Significato
DI	Digital Input
DO	Digital Output
AI	Analog Input
AO	Analog Output
RT	Real Time
H24	Always active
MASK	Mascherato
LEN	Lunghezza impulso

Tipi di Automazioni

Tipo	Uso
simple	PIR, porte, finestre
trend	umidità, temperatura
debounce	segnali rumorosi
composite	logiche complesse
bitmask	comandi manuali
scheduled	cron
dynamic	ogni X ms

Glossario Tecnico

- **Field** → valore letto da Modbus
- **ToPanel** → valore scritto verso pannello
- **FromPanel** → valore letto dal pannello
- **VirtualArea** → area non mappata a dispositivo
- **Cooldown** → pausa dopo errori Modbus
- **Split-Out** → replica temporizzata
- **Toggle** → fronte ON/OFF
- **Scene** → gruppo di azioni

Note di Implementazione

- non usare delay()
- non bloccare mai il loop
- non scrivere nel buffer senza timestamp
- non modificare profili builtin

- definire profili custom in DMSetup.hpp
- usare Watch Areas per debug

Setup dell'IDE Arduino

Di seguito trovi la procedura completa per preparare l'ambiente di sviluppo Arduino in modo corretto, pulito e professionale, così da poter compilare e caricare senza problemi il software DomoManager su Opta o altre board supportate.

01

Installare Arduino IDE

Obbligatorio

Serve per compilare e caricare il firmware DomoManager sulle board.

<https://www.arduino.cc/en/software>

- Scarica **Arduino IDE 2.x** per Windows, macOS o Linux
- Installa con le opzioni predefinite
- Avvia l'IDE per verificare che si apra correttamente, all'apertura l'IDE si auto aggiorna e carica i componenti base.

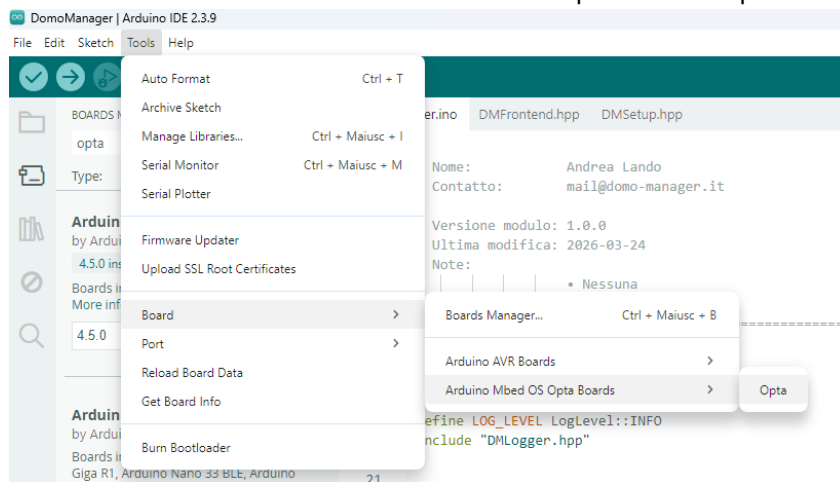
02

Installare le board necessarie (Opta, ESP32, ecc.)

DomoManager richiede che le board siano installate nel Board Manager.

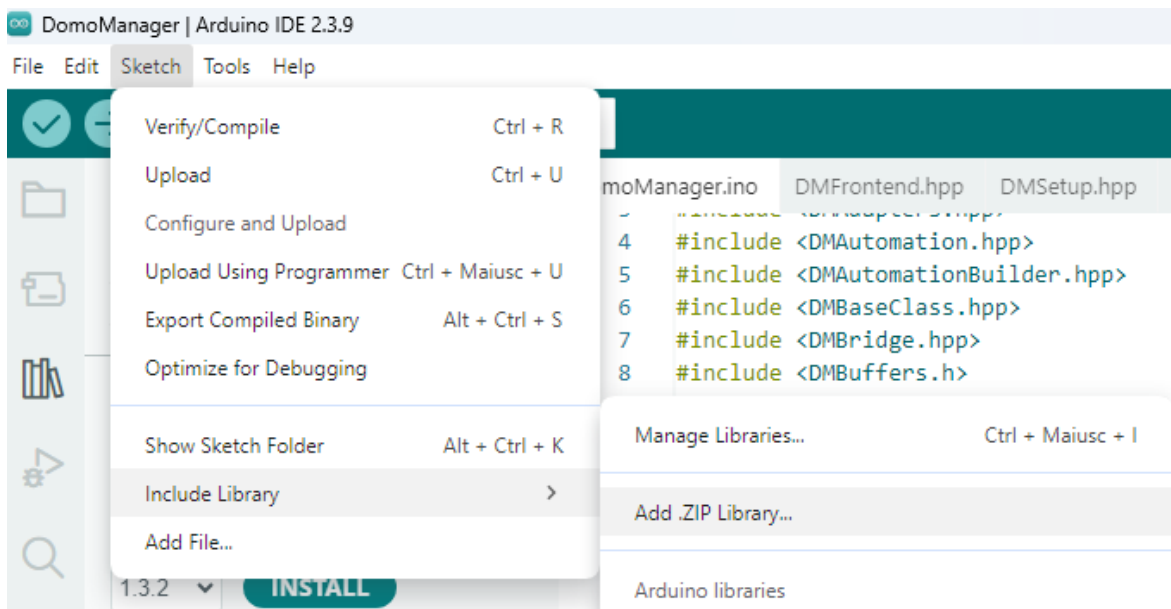
Arduino IDE → **Tools** → **Board** → **Boards Manager**

- Cerca **Arduino Opta** e installa (Il nome della Board completo è **Arduino Mbed OS Opta Boards**, al momento, versione 4.5)
- Cerca **ESP32** (Espressif Systems) e installa
- Attendi la fine del download prima di chiudere la finestra
- Selezionare in Tools->Board->Arduino Mbed OS Opta Boards->Opta



Installare le librerie di DomoManager

Da GitHub scaricare le librerie di DomoManager in formato .zip (nome file **DomoManagerLibs**), copiarlo in una cartella a piacere. Aprire l'IDE Arduino, ed installare le librerie dal menu: Arduino IDE → **Sketch -> Include Library -> Add .ZIP Library**, selezionando il file **DomoManagerLibs.zip** scaricato precedentemente.



Dopo qualche secondo l'IDE visualizza il messaggio "La libreria è stata installata".

Tornando nel menu Arduino IDE → **Sketch -> Include Library**, in fondo alla finestra noterete

Contributed libraries

DomoManager

Installare le librerie richieste da DomoManager, se richiesto dall'IDE, NON installare le librerie dipendenti

Arduino IDE → **Tools → Manage Libraries...**, ricercare ed installare le seguenti librerie:

- Installa **ArduinoRS485**
- Installa **ArduinoModbus**
- Installa **ArduinoJson**
- Installa **PubSubClient**
- Installa **Arduino_UnifiedStorage**
- Installa **Arduino_POSIXStorage**
- Installa **Arduino_USBHostMbed5**
-

Inoltre

- Installa eventuali librerie aggiuntive richieste dal tuo setup (sensori, display, ecc.), se hai personalizzato DomoManager
- Verifica sempre che le librerie siano aggiornate

04

Configurare la porta

Serve per permettere all'IDE di comunicare con la scheda.

Arduino IDE → **Tools** → **Port**

- Seleziona la porta COM corretta (avendo cura di avere precedentemente connesso OPTA al PC mediante cavo USB).
- Se non appare, prova a cambiare cavo USB o porta del PC

05

Importare il progetto DomoManager

Per compilare correttamente il firmware è necessario aprire la cartella del progetto.

- Apri la cartella del software DomoManager
- Carica il file principale **DomoManager.ino** o equivalente
- Verifica che tutte le tab siano presenti (DMSetup.hpp, DMFrontend.hpp, DomoManager.ino)

06

Compilare il firmware

Verifica

Serve per assicurarsi che l'ambiente sia configurato correttamente.

Arduino IDE → **Verify (✓)**

- Attendi la compilazione
- Se compaiono errori, controlla librerie e board installate
- Se tutto è ok, procedi alla personalizzazione dei parametri

07

Personalizzazione dei parametri

In base alla configurazione hardware da gestire, modificare i parametri di:

- DMSetup.hpp
- DomoManager.ino

consulta a questo riguardo i manuali di Onboarding e di Use cases.

N.B. Questa parte può essere saltata e compilata in seguito.

08

Caricare il firmware su Arduino

Ultimo passo: trasferire il firmware sulla board.

Arduino IDE → **Upload (→)**

- Assicurati che la board sia alimentata correttamente

- Non scollegare il cavo durante il caricamento
- Attendi il messaggio **Upload completed**

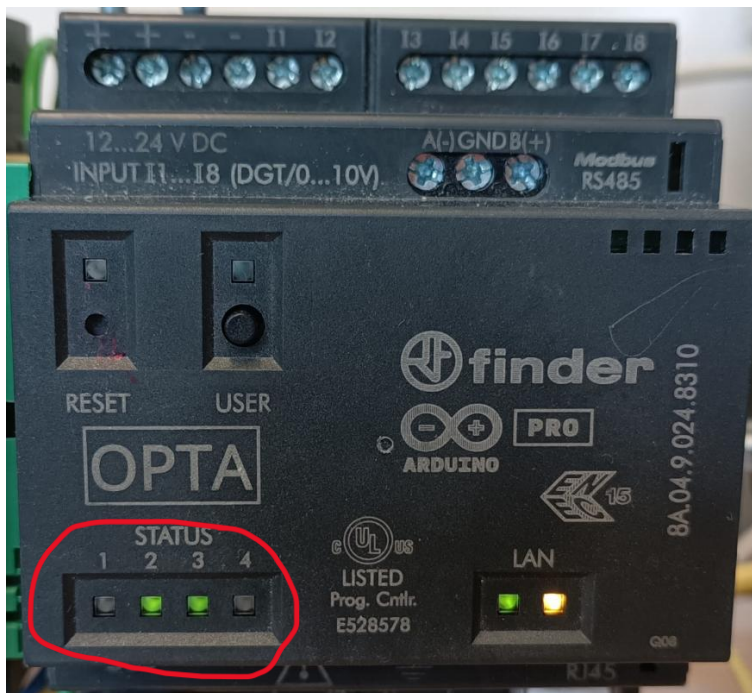
Attenzione: nel caso in cui la configurazione di DmSetup.hpp sia vuota, OPTA visualizza i LED centrali sempre accesi e sul Serial monitor dell'IDE, il seguente messaggio: "CONFIGURAZIONE VUOTA → impossibile avviare il sistema, caricare in DmSetup.hpp una configurazione valida".

```

Output  Serial Monitor X
Message (Enter to send message to 'Opta' on 'COM5')

16:18:18.316 -> [2448 ms] [i INFO] [DomoManagerFrontendEngine::SetupEthernet] Ethernet interface started
16:18:18.316 -> [2449 ms] [i INFO] [DomoManagerFrontendEngine::SetupEthernet] My IP address: 192.168.12.202
16:18:18.316 -> [2449 ms] [i INFO] [DomoManagerFrontendEngine::Setup] Modbus TCP to RTU timeout set 250 mSec.
16:18:18.316 -> [2450 ms] [X ERROR] [DomoManager::setup] CONFIGURAZIONE VUOTA - impossibile avviare il sistema, caricare in DmSetup.hpp una configurazione valida

```



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...*
