**Documentazione Tecnica: Dual Serial Communication Manager**

**Versione:** 1.0.0  
**Data:** Luglio 2025  
**Classificazione:** Documentazione Tecnica

**Panoramica del Sistema**

Il **Dual Serial Communication Manager** è una libreria Python per gestire comunicazioni seriali simultanee su due porte distinte. Il sistema è progettato per applicazioni industriali che richiedono separazione tra flusso dati e comandi di configurazione.

**Caratteristiche Principali**

* **Dual-port architecture:** Gestione simultanea di /dev/ttyUSB0 e /dev/ttyAMA10
* **Thread-safe operations:** Accesso sicuro alle risorse seriali in ambiente multi-thread
* **Real-time data acquisition:** Lettura continua non-blocking con callback personalizzabili
* **Error handling robusto:** Gestione completa degli errori di comunicazione
* **Logging integrato:** Sistema di tracciamento per debugging e monitoring

**Architettura del Sistema**

┌─────────────────────────────────────────────────────────────┐

│ DualSerialManager │

├─────────────────────────────────────────────────────────────┤

│ ┌─────────────────┐ ┌─────────────────┐ │

│ │ DATA PORT │ │ SETUP PORT │ │

│ │ /dev/ttyUSB0 │ │ /dev/ttyAMA10 │ │

│ │ │ │ │ │

│ │ • Streaming │ │ • Configuration │ │

│ │ • Read commands │ │ • Parameters │ │

│ │ • Monitoring │ │ • Calibration │ │

│ └─────────────────┘ └─────────────────┘ │

│ │ │ │

│ ┌─────────────────┐ ┌─────────────────┐ │

│ │ data\_lock │ │ setup\_lock │ │

│ │ (Thread Safety) │ │ (Thread Safety) │ │

│ └─────────────────┘ └─────────────────┘ │

│ │

│ ┌─────────────────────────────────────────────────────┐ │

│ │ Continuous Reading Thread │ │

│ │ • Buffer management │ │

│ │ • Line parsing │ │

│ │ • Callback execution │ │

│ └─────────────────────────────────────────────────────┘ │

└─────────────────────────────────────────────────────────────┘

**Configurazione Standard**

* **Baudrate:** 9600 bps
* **Data bits:** 8
* **Parity:** None
* **Stop bits:** 1
* **Timeout:** 1.0 secondi
* **Encoding:** UTF-8

**API Reference**

**Classe DualSerialManager**

**Costruttore**

DualSerialManager(data\_port='/dev/ttyUSB0', setup\_port='/dev/ttyAMA10',

baudrate=9600, timeout=1.0)

**Parametri:**

* data\_port: Porta per streaming dati (default: /dev/ttyUSB0)
* setup\_port: Porta per configurazioni (default: /dev/ttyAMA10)
* baudrate: Velocità comunicazione in baud
* timeout: Timeout operazioni I/O in secondi

**Metodi di Connessione**

**connect(port\_type: PortType) -> bool** Stabilisce connessione con la porta specificata.

* **Parametri:** port\_type - Tipo porta (DATA\_PORT o SETUP\_PORT)
* **Ritorno:** True se connessione riuscita, False altrimenti
* **Eccezioni:** Gestite internamente, logging automatico

**connect\_all() -> Dict[PortType, bool]** Connette entrambe le porte simultaneamente.

* **Ritorno:** Dizionario con stato connessione per ogni porta

**disconnect(port\_type: PortType) -> bool** Disconnette la porta specificata.

**disconnect\_all() -> None** Disconnette tutte le porte e termina thread di lettura.

**is\_connected(port\_type: PortType) -> bool** Verifica stato connessione porta.

**Metodi di Comunicazione**

**send\_read\_command(command: str) -> Optional[str]** Invia comando di lettura su porta dati.

* **Parametri:** command - Comando da inviare
* **Ritorno:** Risposta ricevuta o None se errore
* **Note:** Aggiunge automaticamente terminatori \r\n

**send\_setup\_command(command: str) -> Optional[str]** Invia comando di configurazione su porta setup.

* **Parametri:** command - Comando di setup da inviare
* **Ritorno:** Risposta ricevuta o None se errore
* **Note:** Timeout esteso per operazioni di configurazione

**Metodi di Acquisizione Continua**

**start\_continuous\_reading(callback: Callable[[str], None] = None) -> None** Avvia acquisizione continua dati.

* **Parametri:** callback - Funzione chiamata per ogni linea ricevuta
* **Comportamento:** Esecuzione in thread separato, non-blocking

**stop\_continuous\_reading() -> None** Termina acquisizione continua dati.

**Metodi di Utilità**

**get\_status() -> Dict** Restituisce stato completo del sistema.

* **Ritorno:** Dizionario con informazioni su connessioni e thread

**Implementazione e Utilizzo**

**Setup Base**

from dual\_serial\_manager import DualSerialManager, PortType

# Inizializzazione con parametri default

manager = DualSerialManager()

# Inizializzazione personalizzata

manager = DualSerialManager(

baudrate=115200,

timeout=2.0

)

**Gestione Connessioni**

# Connessione automatica

connections = manager.connect\_all()

for port\_type, status in connections.items():

print(f"{port\_type.value}: {'Connessa' if status else 'Errore'}")

# Connessione selettiva

if manager.connect(PortType.DATA\_PORT):

print("Porta dati operativa")

**Invio Comandi**

# Comando di lettura

response = manager.send\_read\_command("GET\_TEMPERATURE")

if response:

print(f"Temperatura: {response}")

# Comando di setup

result = manager.send\_setup\_command("SET\_BAUDRATE 115200")

print(f"Configurazione: {result}")

**Acquisizione Continua**

def process\_data(data):

"""Callback per elaborazione dati"""

print(f"Ricevuto: {data}")

# Parsing specifico

if "TEMP:" in data:

temperature = float(data.split(":")[1])

print(f"Temperatura: {temperature}°C")

# Avvio acquisizione

manager.start\_continuous\_reading(process\_data)

# Il programma continua mentre i dati vengono processati in background

**Esempi Applicativi**

**Sistema di Monitoraggio Industriale**

class IndustrialMonitor:

def \_\_init\_\_(self):

self.manager = DualSerialManager(baudrate=115200)

self.data\_buffer = []

def setup\_device(self):

"""Configurazione iniziale dispositivo"""

commands = [

"SET\_SAMPLE\_RATE 100",

"ENABLE\_SENSORS ALL",

"SET\_UNITS METRIC",

"SAVE\_CONFIG"

]

for cmd in commands:

response = self.manager.send\_setup\_command(cmd)

print(f"Setup: {cmd} -> {response}")

def data\_handler(self, data):

"""Gestione dati in tempo reale"""

timestamp = time.time()

self.data\_buffer.append((timestamp, data))

# Parsing multi-sensore

if data.startswith("SENSORS:"):

values = data.split(":")[1].split(",")

temp, humidity, pressure = map(float, values)

# Controlli di soglia

if temp > 75.0:

self.emergency\_shutdown()

def emergency\_shutdown(self):

"""Procedura di emergenza"""

self.manager.send\_setup\_command("EMERGENCY\_STOP")

print("EMERGENZA: Sistema spento")

def run(self):

"""Avvio sistema completo"""

self.manager.connect\_all()

self.setup\_device()

self.manager.start\_continuous\_reading(self.data\_handler)

# Polling periodico

while True:

self.manager.send\_read\_command("GET\_STATUS")

time.sleep(10)

**Sistema di Calibrazione Automatica**

class AutoCalibrator:

def \_\_init\_\_(self):

self.manager = DualSerialManager()

self.calibration\_points = {}

def calibrate\_sensor(self, sensor\_id, reference\_value):

"""Calibrazione singolo sensore"""

# Lettura valore corrente

current = self.manager.send\_read\_command(f"READ\_SENSOR {sensor\_id}")

if not current:

return False

current\_value = float(current.split(":")[1])

offset = reference\_value - current\_value

# Applicazione calibrazione

calib\_cmd = f"CALIBRATE {sensor\_id} {offset}"

result = self.manager.send\_setup\_command(calib\_cmd)

return "OK" in result

def full\_calibration\_sequence(self):

"""Sequenza calibrazione completa"""

sensors = {1: 25.0, 2: 50.0, 3: 1013.25} # ID: valore\_riferimento

results = {}

for sensor\_id, ref\_value in sensors.items():

results[sensor\_id] = self.calibrate\_sensor(sensor\_id, ref\_value)

time.sleep(1) # Stabilizzazione

# Salvataggio permanente

if all(results.values()):

self.manager.send\_setup\_command("SAVE\_CALIBRATION")

print("Calibrazione completata e salvata")

return results

**Error Handling e Debugging**

**Gestione Errori Standard**

Il sistema implementa un approccio a tre livelli per la gestione degli errori:

1. **Livello Hardware:** Eccezioni serial.SerialException
2. **Livello Applicativo:** Eccezioni generiche Python
3. **Livello Logico:** Timeout e risposte malformate

**Logging Configuration**

import logging

# Setup logging personalizzato

logging.basicConfig(

level=logging.INFO,

format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',

handlers=[

logging.FileHandler('serial\_comm.log'),

logging.StreamHandler()

]

)

# Logger specifico per debugging

debug\_logger = logging.getLogger('SerialDebug')

debug\_logger.setLevel(logging.DEBUG)

**Diagnostica delle Connessioni**

def diagnose\_system(manager):

"""Funzione diagnostica completa"""

import os

print("=== DIAGNOSTICA SISTEMA ===")

# Verifica esistenza dispositivi

for port\_type in PortType:

config = manager.ports\_config[port\_type]

port\_path = config['port']

if os.path.exists(port\_path):

print(f" {port\_path}: Dispositivo presente")

else:

print(f" {port\_path}: Dispositivo non trovato")

continue

# Test connessione

if manager.connect(port\_type):

print(f" {port\_path}: Connessione OK")

# Test comunicazione

if port\_type == PortType.DATA\_PORT:

response = manager.send\_read\_command("PING")

if response:

print(f" {port\_path}: Comunicazione OK - {response}")

else:

print(f" {port\_path}: Nessuna risposta")

manager.disconnect(port\_type)

else:

print(f"✗ {port\_path}: Errore connessione")

**Problemi Comuni e Soluzioni**

| **Problema** | **Causa Probabile** | **Soluzione** |
| --- | --- | --- |
| **Connessione fallisce** | Porta occupata o permessi | sudo usermod -a -G dialout $USER |
| **Timeout comandi** | Baudrate errato | Verificare configurazione dispositivo |
| **Dati corrotti** | Interferenze elettriche | Controllare cablaggio e schermatura |
| **Thread bloccato** | Deadlock nei lock | Verificare ordine acquisizione lock |
| **Memory leak** | Callback non ottimizzati | Profilare e ottimizzare callback |

**Performance e Ottimizzazioni**

**Metriche di Performance**

* **Latenza comando:** ~10ms (tipica)
* **Throughput dati:** Fino a 90% del baudrate teorico
* **CPU usage:** <5% su ARM Cortex-A8 @1GHz
* **Memory footprint:** ~2MB RAM

**Best Practices Implementative**

**Thread Safety**

# Corretto: uso dei context manager

with manager.data\_lock:

response = connection.read()

# Scorretto: accesso diretto senza lock

response = connection.read() # Race condition!

**Resource Management**

# Pattern RAII raccomandato

try:

manager = DualSerialManager()

manager.connect\_all()

# ... operazioni ...

finally:

manager.disconnect\_all() # Cleanup garantito

**Callback Optimization**

def optimized\_callback(data):

"""Callback ottimizzato per performance"""

# Evitare operazioni I/O pesanti

# Preferire operazioni in-memory

if data.startswith("CRITICAL:"):

# Processing immediato per dati critici

handle\_critical\_data(data)

else:

# Queue per processing asincrono

data\_queue.put(data)

**Configurazione Sistema Linux**

**Permessi e Accesso**

# Aggiunta utente al gruppo dialout

sudo usermod -a -G dialout $USER

# Verifica porte disponibili

ls -la /dev/tty\*

# Setting permessi specifici

sudo chmod 666 /dev/ttyUSB0

sudo chmod 666 /dev/ttyAMA10

**Ottimizzazioni Kernel**

# Buffer seriali aumentati

echo 4096 > /sys/class/tty/ttyUSB0/rx\_buffer\_size

# Scheduling real-time per thread critici

chrt -f 50 python3 serial\_app.py

**Monitoring e Debugging**

# Monitoring traffico seriale

sudo cat /dev/ttyUSB0 | hexdump -C

# Analisi performance

strace -e trace=read,write python3 serial\_app.py

# Process monitoring

top -p $(pgrep -f serial\_app.py)

**Specifiche Tecniche Hardware**

**Requisiti Minimi Sistema**

* **OS:** Linux kernel 3.0+ (POSIX-compliant)
* **Python:** 3.7+
* **RAM:** 512MB disponibili
* **CPU:** ARM Cortex-A8 o equivalente x86
* **Porte seriali:** 2x UART hardware

**Compatibilità Dispositivi**

* **USB-Serial converters:** FTDI, Prolific, CP210x
* **Embedded UART:** Raspberry Pi, BeagleBone, i.MX6
* **Industrial modules:** Advantech, Moxa, Digi

**Limitazioni Conosciute**

* **Baudrate massimo:** Limitato da hardware (tipicamente 921600 bps)
* **Distanza cavo:** Max 15m per RS-232, 1200m per RS-485
* **Concurrent connections:** Max 2 porte per istanza manager