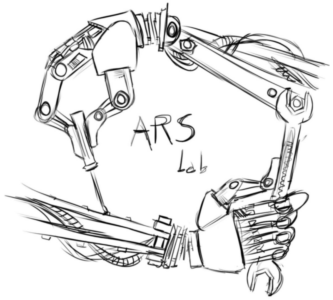
Architettura Software di un Sistema Robotico Autonomo

Corrado Santoro

ARSLAB - Autonomous and Robotic Systems Laboratory
Dipartimento di Matematica e Informatica - Università di Catania, Italy
santoro@dmi.unict.it



Programmazione Sistemi Robotici

Esempi di Archietture Hardware/Software di un Sistema Robotico

- **Architettura monolitica**
  - Unica scheda a MCU (microcontrollore) per il controllo del sistema
  - Driver hardware a bordo della scheda per la gestione dei sensori e degli attuatori
  - Software organizzato con insieme di task concorrenti (sulla base dello schema complessivo del sistema di controllo)
  - Real-time OS
  - Necessità di un MCU performante

Esempi di Archietture Hardware/Software di un Sistema Robotico

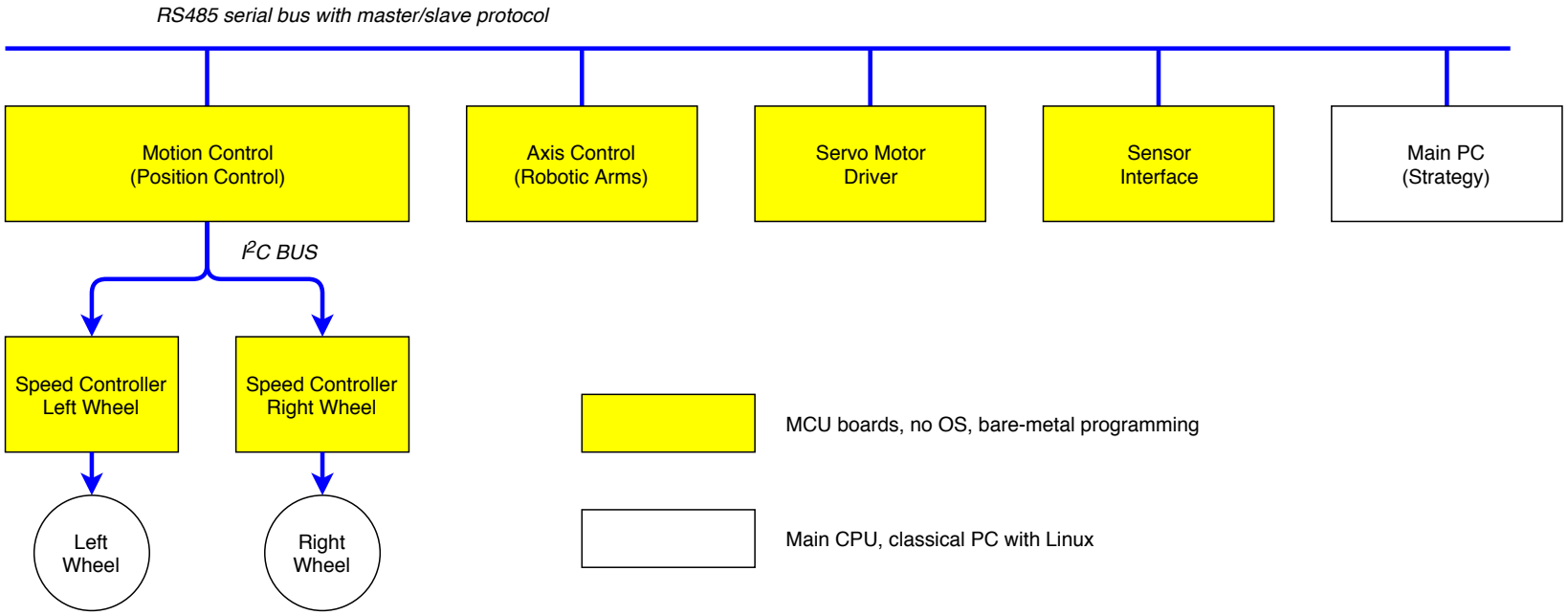
- **Architettura distribuita**
  - Differenti schede a MCU, ognuna specificatamente progettata per un certo specifico controllo (locomozione, servo-motori, bracci robotici, etc.)
  - Bare-metal programming (no SO)
  - Sistema di comunicazione tra schede (RS485, CAN-BUS, SPI, etc.)
  - Eventuale companion PC (anche embedded, es. Raspberry O-Droid, etc.) per gestione della strategia/comportamento del sistema robotico

Esempi di Architetture Hardware/Software di un Sistema Robotico

- **Architettura ibrida**
  - Soluzione monolitica per i task di controllo “critici”
  - Companion PC (anche embedded, es. Raspberry O-Droid, etc.) per gestione della strategia/comportamento del sistema robotico

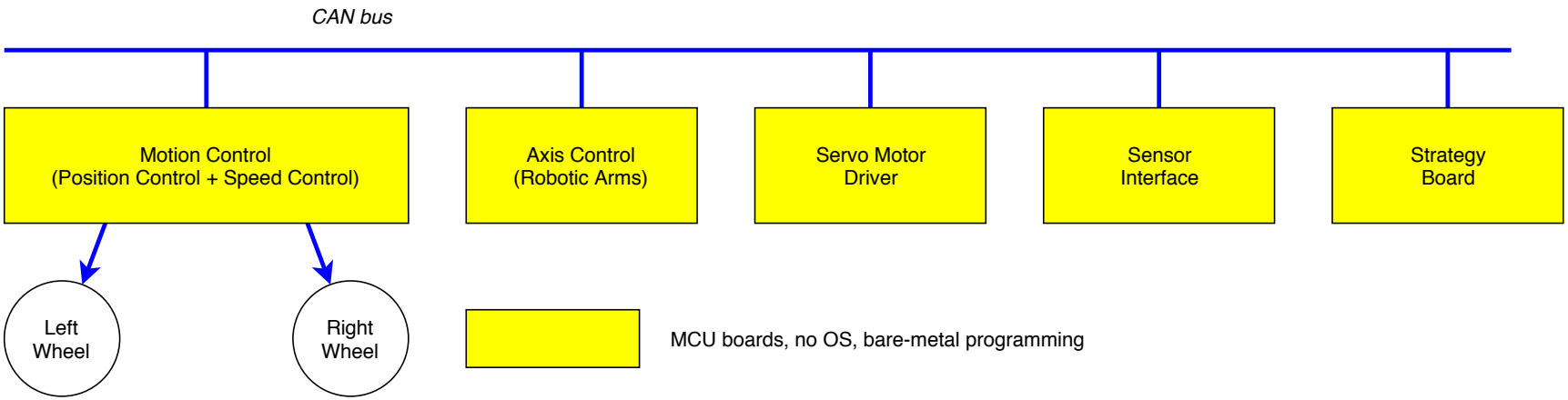
Architettura dei robot per Eurobot 2007-2018

Architettura dei robot per Eurobot 2007-2012



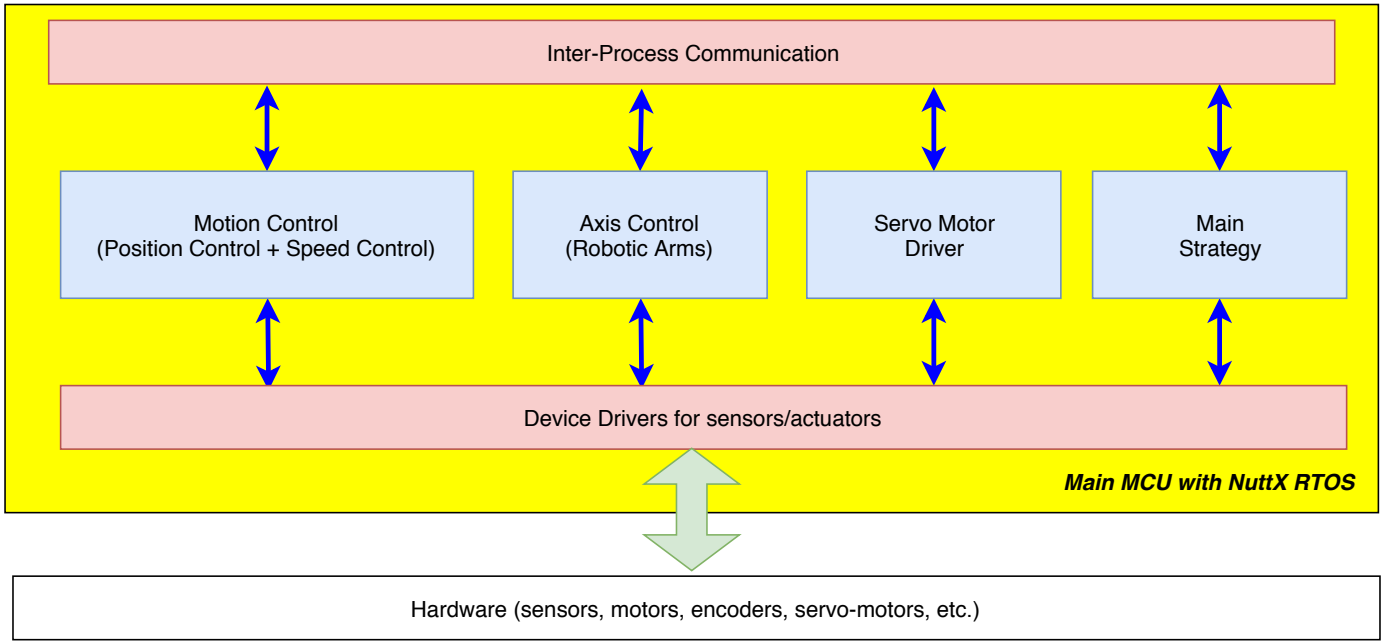
- Architettura distribuita
- Uso di MCU Microchip a 8 bit
- Distribuzione hw/sw sulla base dei sotto-sistemi di controllo presenti
- No RTOS, nessuna necessità di gestire il parallelismo
- Comunicazione attraverso bus RS485 con protocollo master-slave (implementato totalmente in software)
- Control task scritti in C
- Strategia scritta in Erlang (2007-2009) e Python (2010-2012)

Architettura dei robot per Eurobot 2013-2017



- Architettura distribuita
- Uso di MCU Microchip a 16 bit anche per la parte di strategia
- Distribuzione hw/sw sulla base dei sotto-sistemi di controllo presenti
- No RTOS, nessuna necessità di gestire il parallelismo
- Comunicazione attraverso CAN BUS con protocollo gestito interamente dall'hardware
- Control task scritti in C/C++
- Strategia scritta in C/C++

Architettura dei robot per Eurobot 2018



- Architettura monolitica
- Uso di MCU ARM STM32 a 32 bit
- Distribuzione dei task sulla base dei sotto-sistemi di controllo presenti
- NuttX RTOS
- Device drivers per il collegamento con i sensor/attuatori
- Comunicazione tra task attraverso meccanismi di IPC
- Tutto il software scritto in C/C++

RTOS: Architettura di NuttX

NuttX

- **NuttX** è un RTOS multi-piattaforma interamente realizzato da Gregory Nutt
- Il primo rilascio è del 2007 (con licenza BSD)
- Supporta architetture molteplici da 8 a 32 bit
- L'architettura è Unix-like e le chiamate di sistema sono POSIX-compliant
- Sistema di build basato su "gcc" e "qconfig" (stile kernel Linux)
- Footprint di memoria piccolo (es. 250 KBytes sui nostri robot)

- **Memory Model**
  - Flat
  - Protected con supporto da MMU
- **Modello Processi e Scheduling**
  - Pre-emptive scheduling
  - Supporto per processi e thread
  - Scheduling con priorità garantita
  - Politiche di scheduling: FIFO, round-robin
  - Controllo concorrenza POSIX-like (message queues, semafori, mutex, condition variables, etc.)
- **Modello File-System**
  - Device driver (“/dev/” file-system) per moltissime periferiche embedded
  - Supporto per filesystem su SDCard o USB-Flash
  - ROM-FS per file di configurazione, applicazioni, e script di avvio
  - Sequenza di boot Unix-like con script “/etc/rcS”

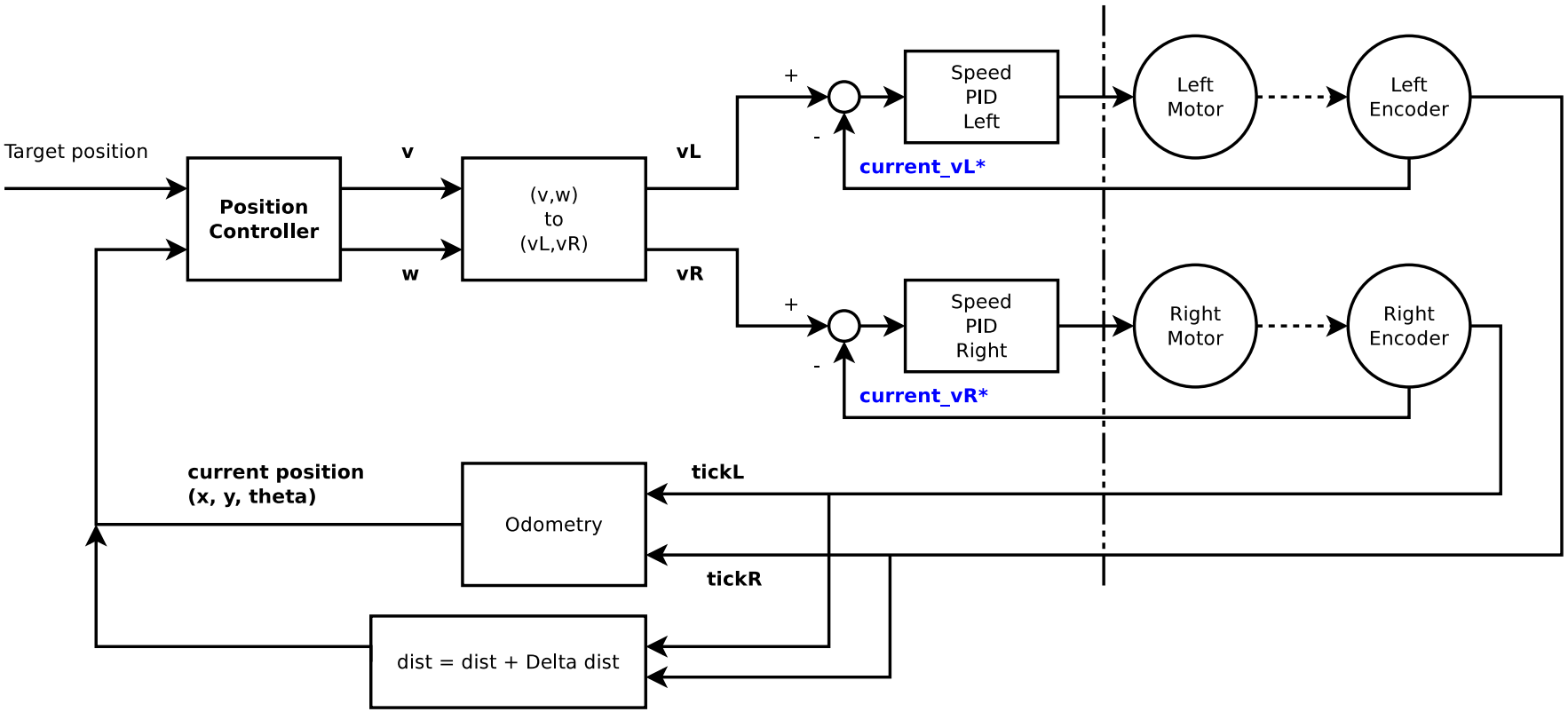
- **User interface**
  - bash-like sheel (nsh) per l'interazione con il sistema
- **Application Model**
  - Applicazioni: "pezzi di codice" stand-alone ognuno con il suo "main"
  - Il build finale generato contiene sempre il codice di tutte le applicazioni
  - Ogni applicazione di sistema è avviata esplicitamente sulla base del file "/etc/rcS", presente nel ROM-filesystemm
  - Qualora un applicazione implementi un loop di controllo, il codice avvia:
    1. Un task "deamon", in background
    2. Le attività periodiche attraverso l'uso di un timer hardware

NuttX Applications - Eurobot 2018

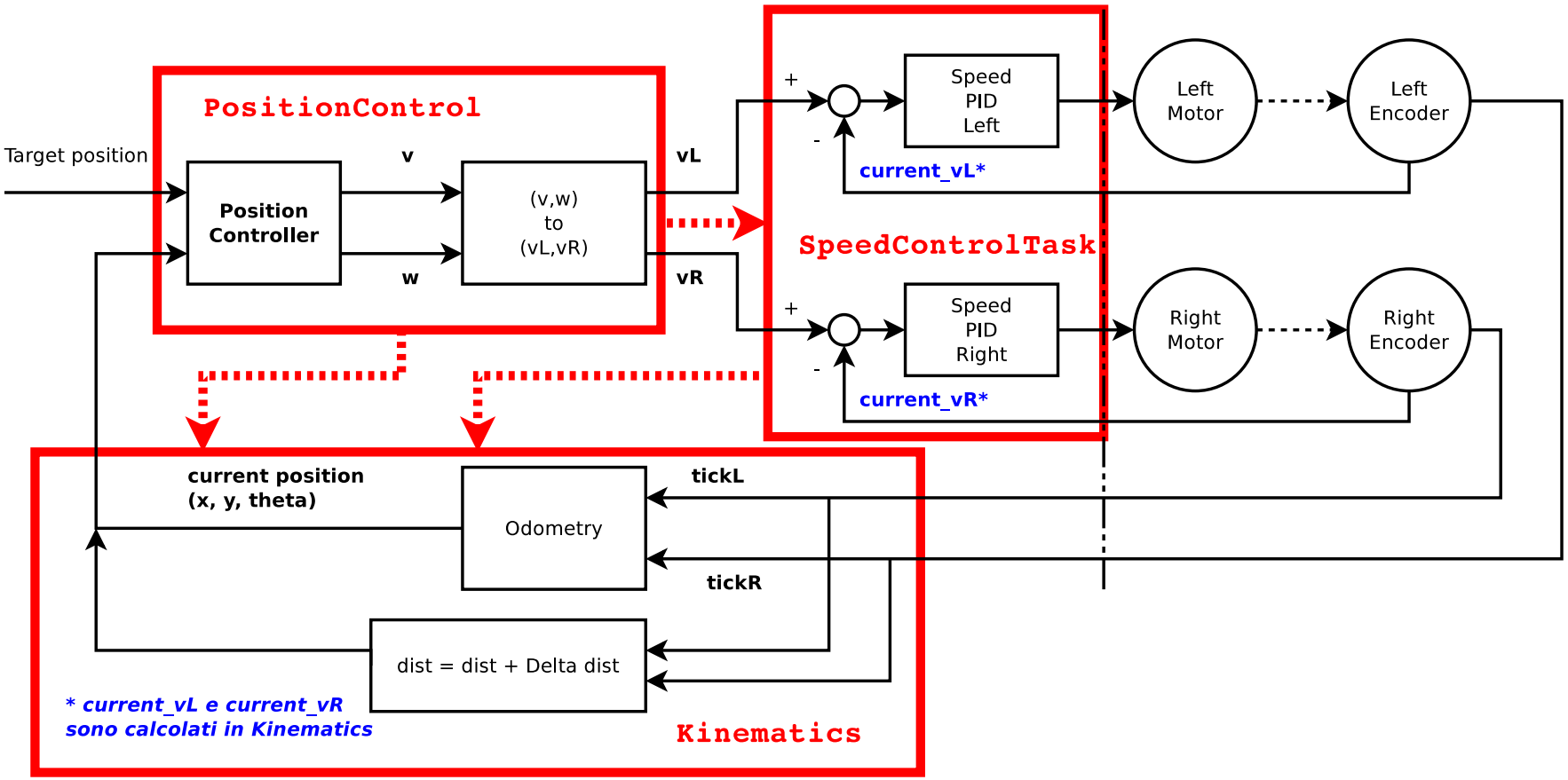
- **`mc:`** motion control
- **`axis:`** arm control (controllo di posizione + velocità di assi singoli)
- **`telemetry:`** app per l'invio wireless dei dati di telemetria
- **`gpio:`** app per la gestione delle linee di I/O digitale
- **`st:`** main strategy

Architettura del Motion Control sui robot per Eurobot

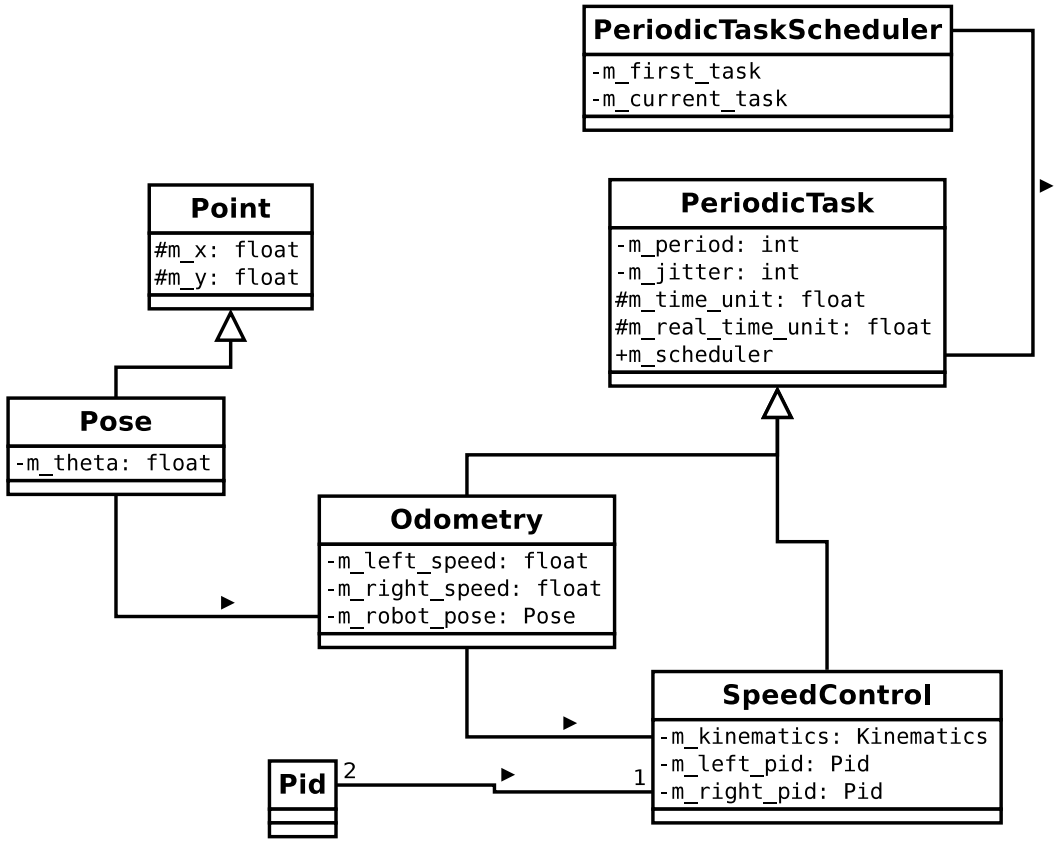
Motion Control: Schema di Controllo



Motion Control: Architettura di base



Motion Control: Class Diagram



- Insieme di **task periodici** ognuno dei quali effettua le attivita di gestione e controllo della movimentazione.
- In più sono presenti alcune classi che rappresentano informazioni aggiuntive sullo stato del robot

Task Periodici

- Il Motion Control possiede un sistema di scheduling (**`PeriodicTaskScheduler`**) per i task periodici di un processo NuttX
- Ogni task periodico è rappresentato da una sottoclasse di **`PeriodicTask`** ed è caratterizzato dal **periodo di esecuzione** dove è stato ridefinito il metodo **`run()`**
- Il **`PeriodicTaskScheduler`** si occupa di eseguire il metodo **`run()`** di ogni PeriodicTask allo scadere del periodo relativo
- Il sistema si basa su un **quanto di tempo** prefissato T_{sched}: **il periodo di ogni task deve essere un multiplo intero di** T_{sched}
- Nel motion control T_{sched} è fissato a $5ms$
- Lo scheduling è **non-preemptive**, il metodo **`run()`** viene eseguito fino alla fine e non può essere interrotto
- Il metodo **`run()`** dovrà pertanto eseguire un singolo "step" del task periodico specifico

Classe PeriodicTask

```
class PeriodicTask {
 public:
    PeriodicTask(PeriodicTaskScheduler & sched, const char * name, int period);
    virtual void run() = 0;
    virtual void on() { m_on = true; };
    virtual void off() { m_on = false; };
    virtual bool on_status() { return m_on; };
    virtual void set_period(int v) { m_period = v; };
 private:
    void execute();
    const char * m_name;
    int m_ticks;
    int m_period;
    bool m_on;
    PeriodicTask * m_task_next;
 protected:
    float m_real_time_period;
    float m_time_unit;
};
```

Classe PeriodicTask

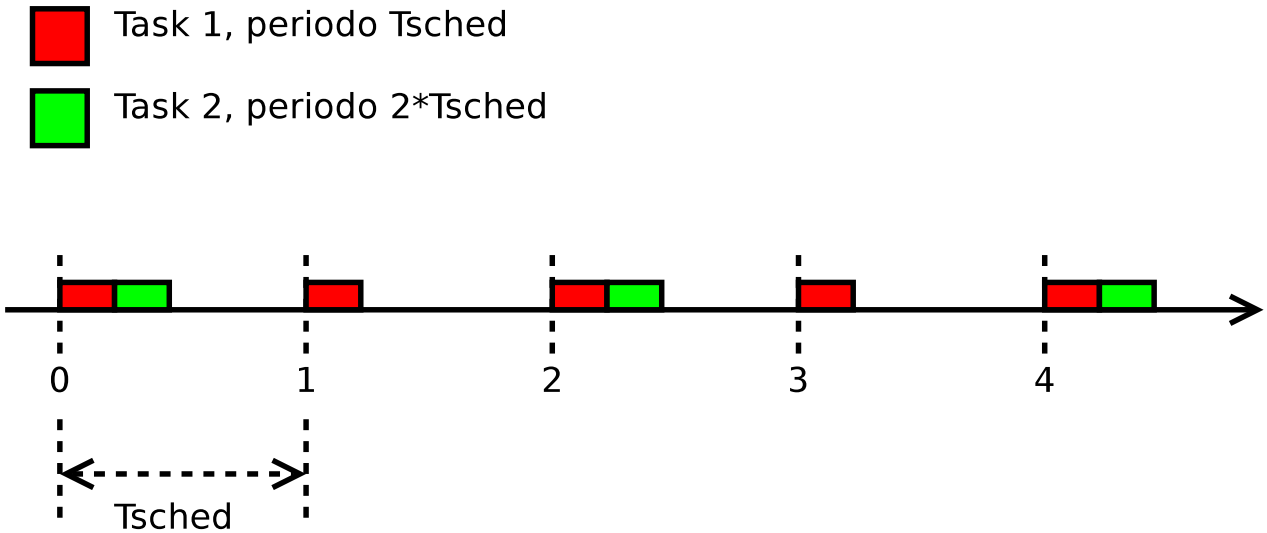
```
class PeriodicTask {
 public:
    PeriodicTask(PeriodicTaskScheduler & sched,
                 const char * name, int period);
...
};
```

Parametri del costruttore:

- **sched**, riferimento allo schedulatore dei task periodici
- **name**, nome del task
- **period**, periodo, in multipli di T_{sched}

Periodo

- Supponiamo di avere due task periodici, con periodi rispettivamente, T_{sched} e $2T_{sched}$
- L'esecuzione avverrà secondo quanto riportato in figura:

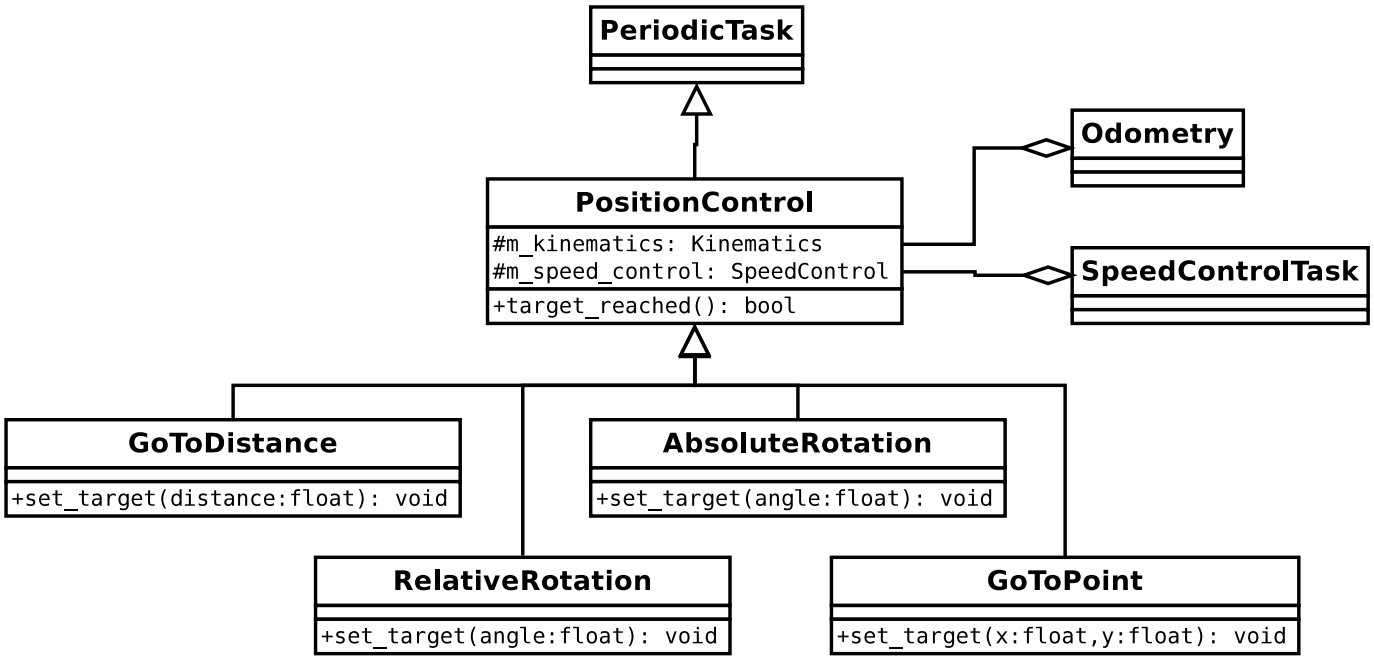


Esecuzione dei PeriodicTask

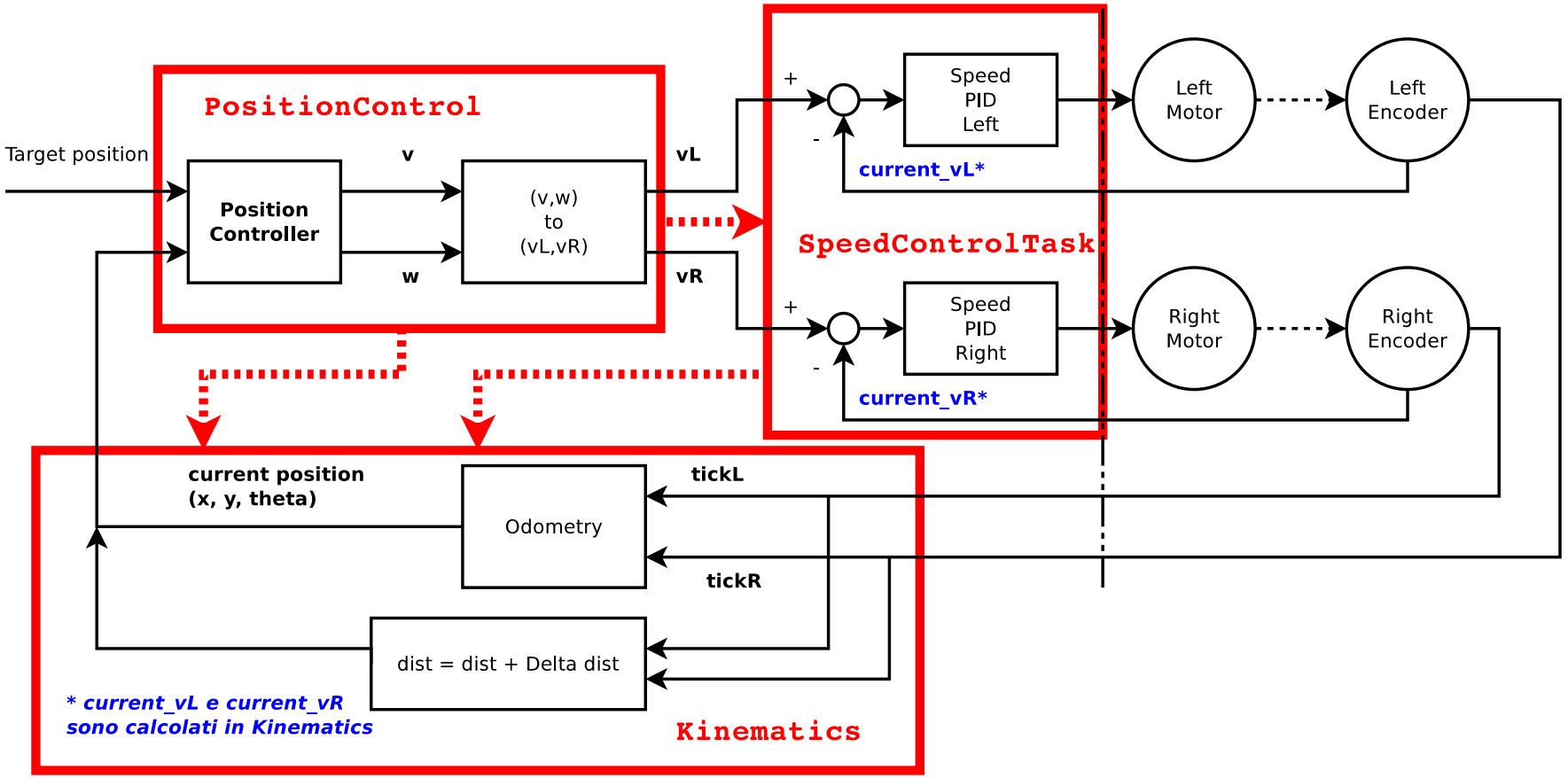
```
PeriodicTaskScheduler mc_scheduler;

static int motion_control_start(int argc, FAR char ** argv)
{
    robot_pose = new Pose();

    // creazione dei task
    odometry = new Odometry(mc_scheduler, *robot_pose,
                            new Encoder("/dev/qe0"), new Encoder("/dev/qe1"),
                            l, r, wb, 4000.0, bumper, alt_bumper);
    wheel_speed = new SpeedControlTask(mc_scheduler, *odometry, *wheels);
    position_control = new PositionControlProxy(mc_scheduler);

    // inserimento dei task ``in ordine'' nella lista di scheduling periodico
    odometry->sched_append();
    position_control->sched_append();
    wheel_speed->sched_append();

    // apertura device timer e configurazione timeout a T_sched
    // ....
    while (true) {
      // attesa del timeout
      if (sigwaitinfo(&mask, nullptr) != SIGALRM)
        continue;
      mc_scheduler.run_all_tasks(t);  // esecuzione dei task
    }
```

Controllo in Posizione: Schema delle classi

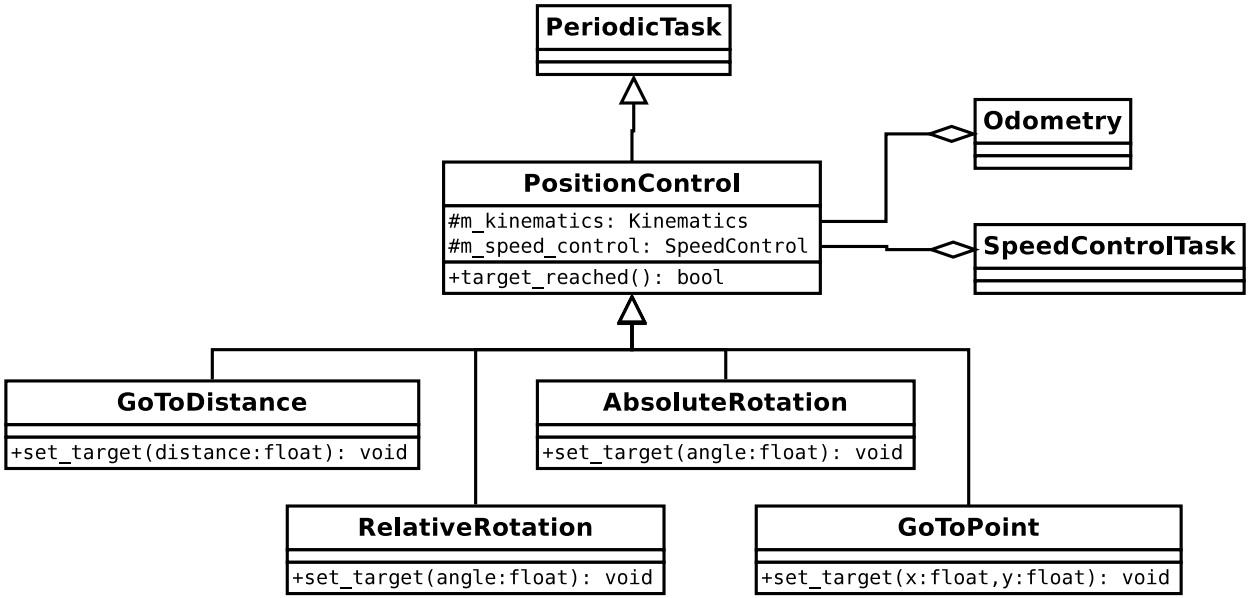
- La classe astratta **`PositionControl`** rappresenta il task periodico che implementa il **generico controllo** in posizione
- Essa include un riferimento all'oggetto **`Odometry`** ed all'oggetto **`SpeedControlTask`**
- Da **`PositionControl`** vengono derivate le classi che implementano lo **specifico controllo**



Relazioni tra Schema di Controllo e Oggetti Software

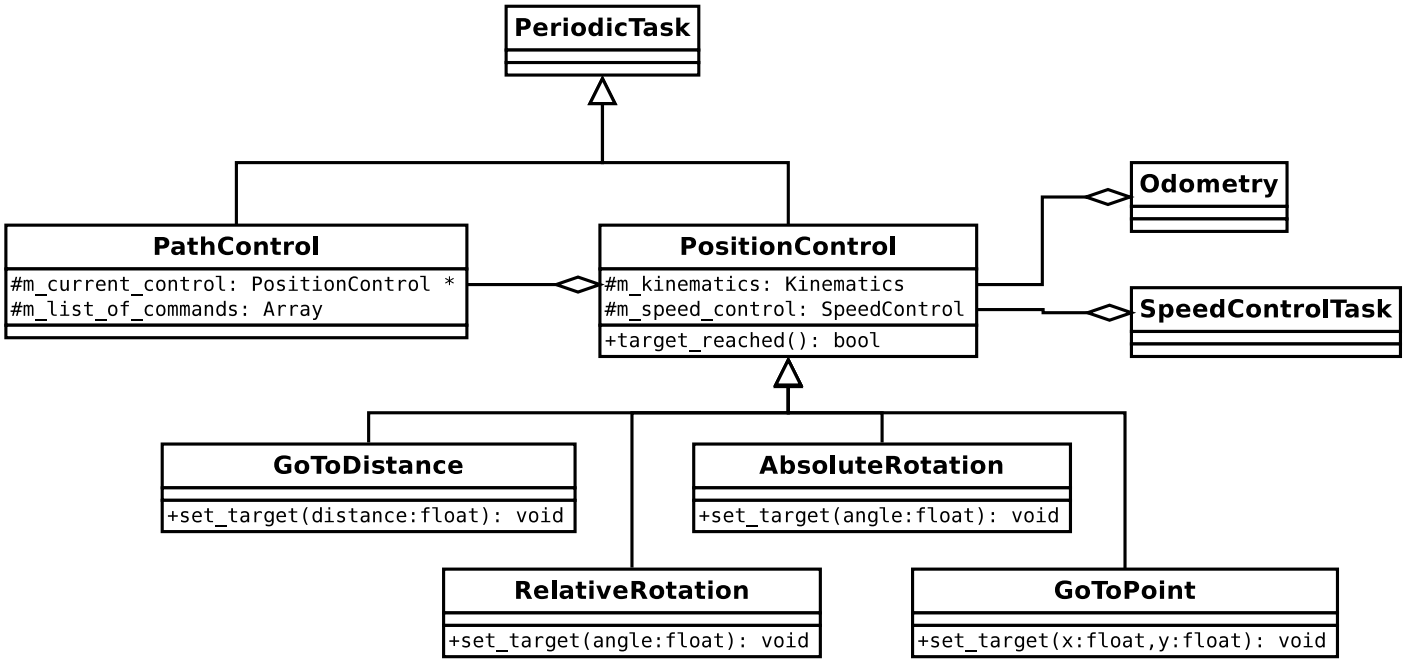


Controllo in Posizione “Intercambiabile”



- Dei vari “`PeriodicTask`” che implementano gli specifici controlli in posizione, **solo uno per volta** può essere attivo
- Ciò viene realizzato andando ad “accendere”/”spegnere” i singoli task periodici
- Ogni task periodico può essere infatti acceso o spento con:
  - **metodo** `void on()`, accende il task
  - **metodo** `void off()`, spegne il task (esso non verrà più schedulato)

Path Control

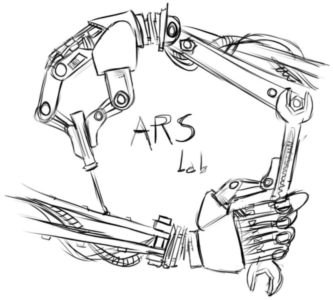


- La classe **`PathControl`** gestisce una **lista di comandi** di geometria
- Usa un puntatore a **PositionControl** che rappresenta il controllore in posizione corrente
- Per cambiare controllo:
  - `m_current_control->off();`
  - `m_current_control = // nuovo controllo;`
  - `m_current_control->set_target(...);`
  - `m_current_control->on();`

Architettura Software di un Sistema Robotico Autonomo

Corrado Santoro

ARSLAB - Autonomous and Robotic Systems Laboratory
Dipartimento di Matematica e Informatica - Università di Catania, Italy
santoro@dmi.unict.it



Programmazione Sistemi Robotici