Tecnologie dei Sistemi di Controllo

Prof. Marcello Bonfè

Dipartimento di Ingegneria

Università di Ferrara

Tel.: +39 0532 974839

E-mail: `marcello.bonfe@unife.it`

Indice

Parte B

Sistemi di Elaborazione e Controllo

Capitolo B.1

Caratteristiche generali dei sistemi per il controllo

Nelle moderne applicazioni di controllo automatico, il dispositivo elettronico che realizza i compiti di regolazione è tipicamente costituito da un sistema *digitale*, cioè basato su un microprocessore opportunamente programmato per eseguire ciclicamente l'algoritmo di controllo e dotato di specifiche periferiche per l'acquisizione di segnali da sensori/trasduttori ed il comando dei convertitori di potenza e degli attuatori in genere. In questa parte delle dispense verranno pertanto analizzate le problematiche di progetto di un sistema a microprocessore per il controllo automatico, evidenziando le caratteristiche peculiari che differenziano i sistemi orientati al controllo rispetto a quelli di utilizzo generico, quali ad esempio quelli per l'elaborazione dati da ufficio.

B.1.1 Funzionalità dei sistemi per il controllo automatico

La funzionalità principale di un sistema a microprocessore per il controllo automatico è quella di interagire con un sistema fisico, da controllare, attraverso sensori ed attuatori. Altre interazioni che il sistema di controllo deve gestire, in genere con esigenze meno prioritarie, sono quelle per lo scambio di informazioni con altri sistemi di elaborazione (es. altri controllori, sistemi per la memorizzazione di dati o *datalogger*, ecc.) o direttamente con operatori umani.

Analizzando anche sommariamente il modo di operare di un sistema per il controllo, si può evidenziare come esso sia completamente diverso rispetto ad un sistema per l'elaborazione dati generico:

a) **un sistema per l'elaborazione dati** accetta dati in ingresso su cui applica l'*algoritmo* o *procedura* per cui è stato programmato, restituisce il risultato dell'elaborazione e **termina l'esecuzione**.

b) **un sistema di elaborazione per il controllo** acquisisce dati dal campo, li elabora secondo l'*algoritmo* o la **logica** di controllo e li attua sul processo da controllare **in modo perpetuo**.

Tale breve descrizione mette comunque in evidenza due fattori fondamentali. Prima di tutto, un sistema per l'elaborazione dati per applicazioni "office", quale ad esempio un Personal Computer (PC), dovendo interagire principalmente con un utente umano è dotato di supporti progettati per permettore all'utente di fornire i dati da elaborare (tastiera, supporti di memoria, ecc.) e di ricevere i risultati (video, stampanti, altri supporti di memoria, ecc.). Inoltre, il tempo richiesto per l'elaborazione NON è vincolato in modo rigoroso a dei valori massimi (non considerando la pazienza dell'utente..) e, in ogni caso, terminato questo tempo il sistema può rimanere in uno stato di attesa fino ad una richiesta successiva.

Il sistema di elaborazione per il controllo, invece, deve essere dotato di dispositivi di **INPUT** (dati ricevuti) e **OUTPUT** (risultati) molto differenti da quelli progettati per l'interazione con umani. Inoltre, siccome il sistema fisico da controllare è in genere dotato di una certa *dinamica*, cioè di una velocità di evoluzione delle sue variabili caratteristiche, la riesecuzione ciclica del programma deve

essere ripetuta secondo una temporizzazione prestabilita (**tempo di campionamento**), da calcolarsi in base alla dinamica del sistema, che rappresenta un vincolo insuperabile per il buon funzionamento del sistema. Pertanto, la corretta esecuzione di un programma per il controllo dipende non solo dalla correttezza logica del software, ma anche dal rispetto di vincoli temporali nella sua esecuzione, che viene pertanto definita come sottoposta al vincolo di **real-time** (*tempo reale*).

Da un lato, sistemi di elaborazione come i PC sono per le applicazioni di automazione, per via dei costi contenuti, della reperibilità di hardware e software e della potenza di calcolo. Dall'altro lato, PC con sistemi operativi per applicazioni "office" non sono progettati per rispettare i vincoli **real-time**. Inoltre, l'affidabilità dei componenti elettronici usati per i comuni PC non garantisce in genere capacità di funzionamento ininterrotto in ambienti ostili e per lunghissimi periodi di tempo (i.e. anni). Pertanto, le scelte relative ai sistemi di elaborazione per il controllo sono orientate verso soluzioni architetturali e tecnologiche differenti, seppure anch'esse standardizzate su determinati tipi di sistemi e componenti.

B.1.1.1 Specifiche architetturali per i sistemi di controllo

La scelta dei componenti tecnologici per il controllo di un impianto industriale o, in generale, di un qualunque sistema ingegneristico, viene influenzata da diversi fattori. Tra i principali si possono citare:

- **La complessità del sistema di controllo**, che può essere misurata in termini di numeri di ingressi ed uscite (sensori ed attuatori) necessari per eseguire le regolazioni automatiche secondo le specifiche funzionali fornite. Nel caso di sistemi molto complessi si applica in genere una scomposizione gerarchica del controllo, organizzandone i componenti in modo da ripartire la necessaria potenza di calcolo ed effettuare scelte tecnologiche più specifiche per alcuni sottosistemi.
- **La distribuzione spaziale**: l'architettura di controllo è condizionata dal fatto che l'impianto da controllare sia localizzato, ad esempio una singola macchina per lavorazioni meccaniche, oppure distribuito all'interno di uno spazio più ampio, quale un grande capannone industriale o addirittura un territorio più vasto (es. il controllo dei sistemi di distribuzione dell'energia elettrica). Nel caso di sistemi così distribuiti occorrerà valutare come decentralizzare i sistemi di controllo e come gestire le comunicazioni tra i livelli locali ed un supervisore generale.
- **La dinamica dei sistemi da controllare**: le specifiche sul tempo di campionamento pongono un vincolo anche sulla potenza di calcolo richiesta all'elaboratore di controllo, solitamente misurata in base al numero di istruzioni eseguite nell'unità di tempo. In base alla dinamica del processo occorrerà prevedere una architettura di controllo di adeguata velocità di calcolo.
- **L'affidabilità necessaria**: in alcuni contesti, le conseguenze di un riavvio del sistema di elaborazione per il controllo, a seguito di un guasto dell'hardware o semplicemente di un errore di esecuzione software, possono essere inaccettabili (es. impianti di produzione energetica, pilota automatico di un aereo commerciale). In tal caso, il sistema di controllo deve essere realizzato con componenti ridondati e che garantiscano bassissime probabilità di guasto, oltre che testato in modo approfondito per verificare la totale assenza di errori software.
- **Il costo**: quest'ultimo rappresenta molto spesso anche il parametro più importante, sebbene i vincoli su di esso richiedano scelte di compromesso nei confronti dei parametri precedenti.

▽ *Esempio: Controllo di un sistema robotizzato.*

Un esempio di sisema di controllo con struttura organizzata in modo gerarchico e funzionalmente distribuita è costituito dala cella robotizzata di Figura B.1.1.

Il sistema di controllo è suddiviso in quattro livelli:

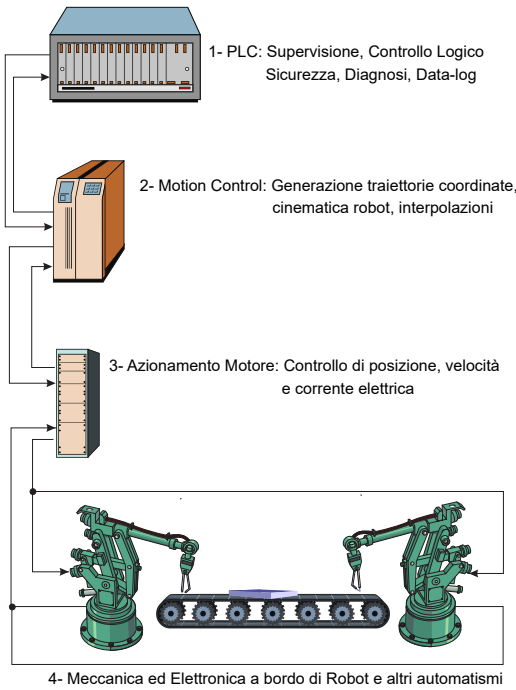


Figura B.1.1: Schema del controllo di una cella di lavoro robotizzata.

1. Il primo livello consiste nel sistema di controllo logico, nel quale sono programmate tutte le sequenze operative descritte tramite stati di elaborazione (es. muovi Robot 1 nel punto A dello spazio Cartesiano, attendi lo spostamento del pezzo sul nastro trasportatore nella postazione Robot 2). Normalmente, nell'Automazione Industriale, tale sistema è costituito da un *Programmable Logic Controller, PLC*.

2. Il secondo livello è costituito dal "controllo del moto" (*Motion Control*), vale a dire l'elaborazione di traiettorie geometriche che realizzino in modo ottimale il comando di movimentazione impartito dal livello precedente. Il dispositivo che realizza questo compito necessita in genere di potenza di calcolo più elevata di quella richiesta al livello precedente, a causa della complessità numerica del calcolo delle traiettorie (es. interpolazioni, cinematica dei robot, ecc.), e viene quindi progettato in maniera specifica.

3. Il livello successivo è costituito dagli azionamenti che si occupano di controllare i motori elettrici che eseguono materialmente la movimentazione, assicurando che la traiettoria calcolata dal livello precedente sia inseguita in modo corretto. Ciascun azionamento viene in genere progettato per controllare una specifica tipologia di motore (DC, Brushless AC, Asincrono, Stepper, ecc.) e tarato per una ancora più specifica taglia di potenza.

4. L'ultimo livello è costituito dalla interfaccia con il processo produttivo, in termini di sensori (posizione, velocità, corrente per i motori dei robot, ma anche fotocellule e sensori di prossimità per il controllo logico dell'impianto) ed attuatori (motori, elettrovalvole pneumatiche, ecc.).

Caratteristica importante di un sistema gerarchico è che ogni livello scambia con quelli adiacenti soltanto le informazioni necessarie e rilevanti al corretto funzionamento della parte interessata.

Ad esempio il livello 1 segnala l'azione da compiere codificata con informazioni logiche (es: abilitazione del movimento del robot 1 verso il punto A), attendendo un segnale di termine del movimento dal livello 2. Il livello 2 genera la traiettoria necessaria per eseguire il movimento e fornisce i riferimenti di posizione (e/o velocità) agli azionamenti dei motori elettrici. Gli azionamenti (livello 3)

eseguono un controllo di posizione in anello chiuso sui motori del robot, avendo come informazione la corrente che scorre sugli avvolgimenti del motore e la posizione del motore stesso (livello 4).

In tale struttura ciascuna sottoparte utilizza un sottoinsieme dei segnali totali che possono essere acquisiti dai sensori della parte operativa (ad esempio al controllo assi occorre l'informazione sulla posizione del motore, ma non la corrente che scorre sugli avvolgimenti elettrici) e quindi non occorre propagare l'intera informazione sensoriale a tutta l'architettura, ottenendo un evidente risparmio in termini di connessioni.

Un'altra considerazione importante riguarda i differenti vincoli temporali a cui ciascun livello deve sottostare. In genere, i vincoli meno stringenti in termini di velocità di elaborazione corrispondono al livello di coordinazione (livello 1), in cui i tempi di reazione richiesti sono dell'ordine di pochi secondi, pertanto è sufficiente che il periodo della ripetizione ciclica del programma sia dell'ordine delle decine o centinaia di millisecondi .

Man mano che scendiamo di livello troviamo tempi sempre più stretti, ad esempio il sistema di Motion Control dovrà lavorare con tempi dell'ordine di pochi millisecondi (tipica dinamica meccanica a velocità elevate), mentre il controllo dei motori elettrici dovrà essere eseguito con tempo di campionamento dell'ordine delle poche centinaia di microsecondi (tipica dinamica delle correnti elettriche).
△

B.1.2 Classificazione dei sistemi di controllo a microprocessore

Come detto, un sistema di elaborazione progettato per eseguire compiti di controllo deve avere determinate caratteristiche. La configurazione minima sarà così composta:

- **Un sistema a microprocessore**, con CPU, memoria, dispositivi di temporizzazione (timer), dispositivi di interfaccia con altri sistemi remoti (reti di comunicazione) o con l'utente (tastiera, schermo).
- **Un sistema di acquisizione dati dal campo**, in grado di acquisire segnali analogici o digitali (codificati e logici) a seconda della applicazione di controllo.
- **Un sistema di attuazione dei segnali di controllo**, comprendente i dispositivi (es. convertitori di potenza) per l'interfacciamento con motori elettrici o attuatori pneumatici operanti sul campo.

I componenti del sistema di controllo elencati possono essere:

- concentrati su di una unica scheda elettronica (**sistemi integrati o *embedded***)
- suddivisi su più schede, collegate fra di loro da un *bus parallelo* ad alta velocità (**sistemi *modulari***).

Il primo sistema è più semplice e, grazie alla sua integrazione, più compatto ed ottimizzato, mentre il secondo è più complesso, ma risulta essere intrinsecamente espandibile grazie alla sua caratteristica modulare (si veda la Tab. B.1.1 per un riassunto delle caratteristiche dei due sistemi).

I sistemi integrati (*embedded*) possono essere progettati in modo ottimizzato, dal punto di vista della compattezza fisica (anche miniaturizzazione), delle caratteristiche dei dispositivi scelti (capacità di calcolo, dissipazione di potenza, ecc.) e delle loro interazioni. Per questi motivi, i sistemi *embedded* sono di solito realizzati ad hoc per svolgere uno specifico compito di controllo, come ad esempio la regolazione di corrente/velocità/posizione per una tipologia di motore elettrico, e vengono in tal caso anche definiti **custom o *special purpose***. Il progettista del sistema di controllo deve quindi sviluppare l'intero sistema (hardware e software) in base alle specifiche richieste, con evidente impegno di tempo e risorse di laboratorio.

Questo tipo di operazione risulta conveniente solamente quando:

- il sistema di controllo deve essere alloggiato in uno spazio limitato (es. centralina elettronica per uso automobilistico);
- il complesso "sistema controllato + controllore" verrà prodotto in serie in notevole quantità (il costo di progettazione viene ripartito su ogni unità prodotta).

Per contro, un sistema di elaborazione custom non può essere adattato ad altri sistemi da controllare diversi da quello per il quale è stato progettato. I sistemi componibili grazie ad un *bus parallelo*, invece, non sono progettati per il controllo di uno specifico sistema, ma hanno caratteristiche generali che consentono di adattarsi a diverse problematiche di controllo. In genere questi sistemi sono composti da uno o più moduli dotati di microprocessore e da schede di espansione per **ingressi e uscite**.

Le schede che compongono i sistema modulari vengono progettate e commercializzate da produttori specializzati, in funzione del tipo di *bus* con il quale queste schede verranno interconnesse. Il progettista del sistema di controllo si limita a scegliere ed assemblare i componenti necessari per il processo complessivo, in base anche alle specifiche richieste. Lo sforzo progettuale si concentra quindi nella programmazione del software che realizza le operazioni richieste dalle specifiche funzionali del processo, operazione non necessariamente di minore difficoltà, ma certamente meno esigente in termini di tempo.

	Sistemi *embedded*	Sistemi *modulari*
Pro	Ottimizzazione del sistema di controllo Elevate prestazioni	Adattabili a diversi tipi di controllo Costi di sviluppo contenuti (solo software)
Contro	Costi elevati di progettazione (software ed hardware) Non adattabili a diversi progetti	Non particolarmente ottimizzati
Utilizzo	Dedicati al controllo locale di un sottosistema	Controllo globale della parte operativa.

Tabella B.1.1: Schema riassuntivo delle prestazioni dei sistemi *embedded* e dei sistemi modulari.

Capitolo B.2

Sistemi *Embedded* per il Controllo

B.2.1 Sistemi a microprocessore dedicati (*embedded*)

I dispositivi di elaborazione progettati in modo dedicato al controllo di una categoria di sistemi fisici devono essere costituiti da un certo numero di componenti fondamentali per la realizzazione del controllo. Tali componenti sono:

1. Un **microprocessore**, con potenza di calcolo e capacità di rappresentazione numerica (N bit per dato[1]) adeguate alla dinamica del sistema fisico e alla precisione richiesta per il suo controllo.

2. Banchi di memoria **RAM**, per contenere le informazioni volatili (i.e. che possono essere cancellate al reset del dispositivo e che costituiscono la cosiddetta **Data Memory**) e **ROM**, che deve invece contenere le informazioni non volatili, principalmente le istruzioni del software di controllo (la cosiddetta **Program Memory**) e le configurazioni del sistema per l'interfacciamento tra i vari dispositivi. [2].

3. Convertitori **A/D e D/A**, per l'acquisizione delle informazione sensoriali e l'attuazione dei comandi agli attuatori (Ingressi e Uscite analogiche), eventualmente preceduti da Multiplexer.

4. Ingressi e Uscite (General Purpose Input/Output, **GPIO**) digitali, sia quelle minime necessarie per il funzionamento del sistema (reset, enable, ecc.), sia quelle di interfacciamento con l'utente (pulsanti e/o display) e con il processo, se il controllo realizzato è di tipo logico.

5. Circuiti di **condizionamento** ed **isolamento** di ingressi e uscite.

6. Dispositivi per il conteggio, che possono essere utilizzati per intercettare determinati eventi (es. legati ad impulsi provenienti da sensori logici), nel qual caso si definiscono **Counters**, o per rilevare lo scorrere del tempo intercettando impulsi provenienti da segnali di clock interni, nel qual caso si definiscono **Timers**. Ovviamente, per un sistema di controllo digitale quest'ultima operazione è di fondamentale importanza per rispettare i vincoli *real-time*.

7. Interfacce di **comunicazione**, come ad esempio un gestore di porte seriali (es. RS-232), per le interazioni con altri componenti "intelligenti" (es. terminale di programmazione, altri dispositivi di controllo, pannelli di interfaccia utente, ecc.).

Tutti questi componenti elettronici possono essere costituiti da chip fisicamente separati tra loro, nel qual caso dovranno quindi essere integrati su una scheda a circuito stampato (**Printed Circuit**

[1] Nel seguito si indicherà con N bit, associati alla classificazione delle CPU, la dimensione dei dati eleborabili in una singola istruzione macchina dalla CPU stessa. Pertanto, CPU ad 8 bit possono elaborare solo singoli byte per istruzione, CPU a 16 bit dati a due byte (word) e così via.

[2] Le parti di codice programmate in modo specifico per la gestione di dispositivi hardware e memorizzate in modo non volatile vengono chiamate **firmware**, in relazione al fatto che esse non sono normalmente flessibili e adattabili da parte dell'utilizzatore finale della scheda ("firm", stabile o permanente)

Board, PCB), oppure, raggiungendo il massimo livello di integrazione possibile, realizzati sullo stesso chip, sfruttando le capacità di miniaturizzazione offerte dai materiali semiconduttori.

La terminologia per indicare dispositivi di elaborazione integrati "on-chip" deve quindi essere estesa a definizioni più specifiche, che evidenziano l'orientamento alla progettazione di sistemi di controllo di tali componenti. In particolare, si possono distinguere, in base alla diversa architettura interna, le seguenti tipologie di processori orientati al controllo:

- **Microcontrollori**
- **Digital Signal Processor (DSP)**

I microcontrollori rappresentano in genere l'evoluzione di un microprocessore di uso comune (*"general purpose"*) nella progettazione dei sistemi di elaborazione, affiancando al nucleo (*"core"*) del dispositivo alcuni o tutti i componenti precedentemente citati. I DSP, invece, sono dei processori caratterizzati da un'architettura interna, significativamente diversa da quella dei processori "general purpose", progettata in modo specifico per realizzare dispositivi per l'elaborazione (es. filtraggio) di segnali analogici digitalizzati. Inizialmente, i processori di tipo DSP si diffusero maggiormente nel contesto dei sistemi di trasmissione ed elaborazione audio e video. Con la loro evoluzione, è stata riconosciuta l'efficienza dei DSP anche nella realizzazione di algoritmi di controllo, che richiedono l'esecuzione ripetitiva di moltiplicazioni e addizioni (**funzioni di trasferimento discretizzate**), così come avviene nelle operazioni di filtraggio ed elaborazione di segnali nelle applicazioni audio/video. Negli ultimi anni, per evidenziare ulteriormente tra i loro prodotti la classe di processori indirizzata in modo specifico alle applicazioni di controllo, molti produttori di circuiti integrati utilizzano il termine **Digital Signal Controller (DSC)** per indicare dispositivi con funzionalità di elaborazione da DSP e caratteristiche di semplicità d'uso e versatilità da microcontrollore.

Nel seguito, verrano evidenziate le differenze architetturali tra microcontrollori e DSP/DSC e le loro applicazioni nel campo del controllo.

B.2.1.1 Microcontrollori ed applicazioni

I microcontrollori costituiscono l'estensione di microprocessori standard, cioè con un nucleo base con CPU, bus controller, interrupt controller e poco altro, con l'integrazione "on-chip" dei dispositivi tipicamente necessari nelle applicazioni di controllo e real-time. fine di ottimizzare i costi e gli ingombri di un sistema di controllo custom. Questi dispositivi mantengono quindi la compatibilità software (perlomeno a livello di linguaggio *Assembly*) con i microprocessori "general purpose" da cui sono derivati, con il conseguente notevole vantaggio di permettere ai progettisti di poter riutilizzare codice esistente e, soprattutto, di limitare al minimo le necessità di formazione tecnica per la conoscenza del dispositivo.

Le Figure B.2.1 e B.2.2, che rappresentano schematicamente l'architettura interna di un generico microprocessore e di un microcontrollore da esso derivato, evidenziano il livello di integrazione di quest'ultimo.

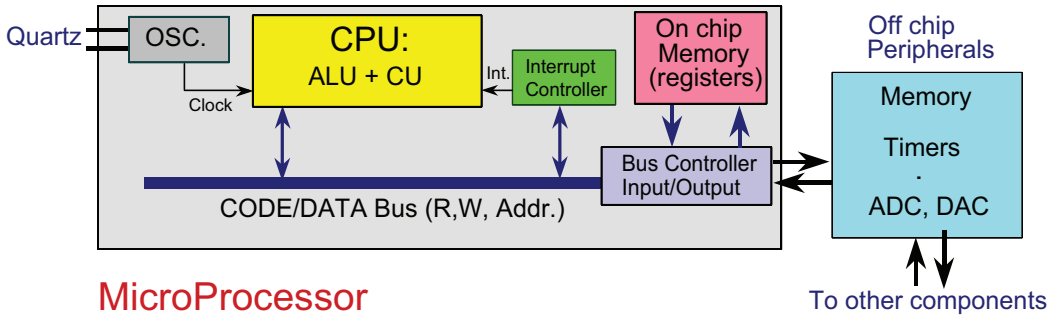


Figura B.2.1: Schema a blocchi di un microprocessore generico ed interfacciamento con periferiche per il controllo

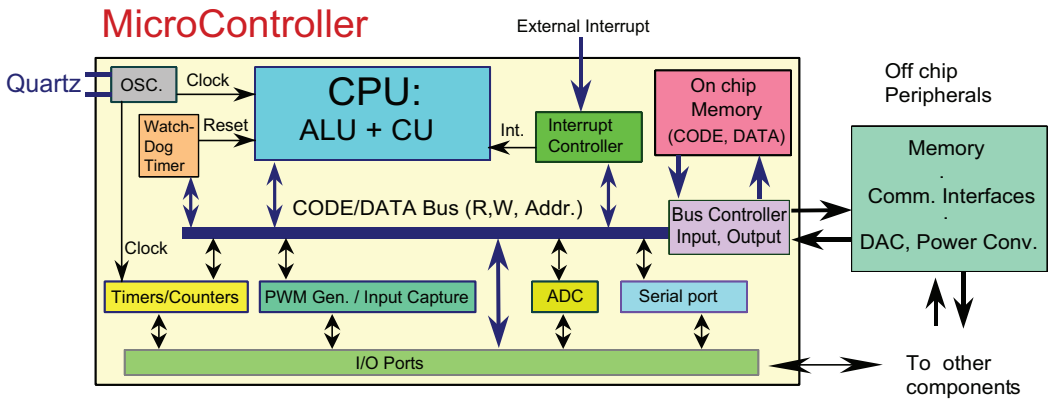


Figura B.2.2: Schema a blocchi di un tipico microcontrollore

La tecnologia costruttiva tipica dei microprocessori standard, di conseguenza anche di molti microcontrollori "storici", prevede la connessione tra il nucleo di elaborazione vera e propria, la **Central Processing Unit o CPU**[3], e le periferiche di memoria tramite un'unica linea di segnali (*data bus*), sulla quale dovranno quindi essere forzatamente scambiti tra CPU e memoria sia le **istruzioni del programma**, sia i **dati memorizzati**, oltre ovviamente agli *indirizzi* di entrambe le informazioni. Si noti pertanto, nelle Figure B.2.1 e B.2.2, che il bus di sistema è appunto riferito come **CODE/DATA bus**. La dimensione del bus di sistema definisce una fondamentale classificazione di microprocessori e microcontrollori (8, 16 o 32 bit). Inoltre, la struttura stessa del bus di sistema definisce l'architettura dell'elaboratore. In particolare, l'architettura che prevede un bus unico per istruzioni e dati viene chiamata architettura di **Von Neumann**, mentre le architetture che prevedono la separazione tra istruzioni e dati, sia dal punto di vista dell'allocazione fisica delle informazioni (banchi di memoria distinti) che dal punto di vista del bus di sistema, vengono chiamate di tipo **Harvard**.

La notevole possibilità di riduzione di ingombro e miniaturizzazione per le schede realizzate con microcontrollori, i cui chip contengono già quasi tutto il "necessario", rende tali dispositivi pressochè onnipresenti negli oggetti di uso quotidiano. Può essere addirittura sorprendente elencare tutte le applicazioni nelle quali vengono utilizzati microntrollori:

- se un elettrodomestico (lavatrice, aspirapolvere, condizionatore d'aria, ecc.) ha un display o una tastiera e usa motori elettrici a velocità variabile contiene certamente un microcontrollore;
- una apparecchiatura per ufficio (stampante, plotter, ecc.), chi richiede diverse movimentazioni di piccoli motori elettrici, contiene un microcontrollore;
- un telecomando ad infrarossi o a radiofrequenza, contiene un microcontrollore per la gestione della modulazione dei dati;
- una moderna automobile contiene decine di microcontrollori, interagenti tramite reti di comunicazione, per regolare l'iniezione carburante (**Electronic Fuel Injection**, EFI), gestire la frenata (**Anti-Blockage System** o ABS), visualizzare informazioni sul cruscotto, gestire apertura/chiusura dei finestrini o i movimenti degli specchi retrovisori esterni, come mostrato dalla Figura B.2.3.

▽ *Esempio: Controllo di un elettrodomestico*

I sensori ed attuatori che si possono trovare all'interno di un comune elettrodomestico come una lavatrice sono quelli necessari alla misura della temperatura e della "durezza" (presenza di calcare

[3] a sua volta costituita dal blocco di elaborazione aritmetica **Arithmetic-Logic Unit, ALU** e dal blocco **Control Unit, CU**, che ne regola il funzionamento gestendo caricamento ed esecuzione delle istruzioni

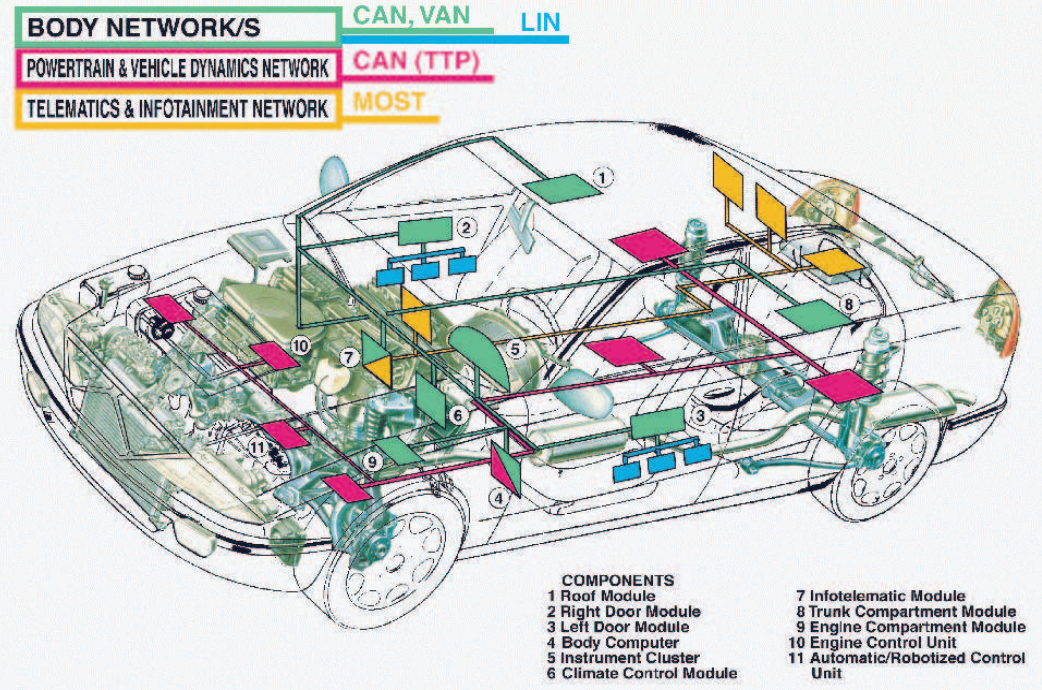


Figura B.2.3: Microcontrollori e reti di comunicazione (CAN, LIN, MOST) nel settore automobilistico

rilevata con misura di conducibilità) dell'acqua, alla movimentazione del cestello con un motore elettrico AC, al riscaldamento ed al pompaggio dell'acqua all'interno del cestello stesso. Sebbene le problematiche di controllo legate a tali dispositivi siano sostanzialmente semplici, non sarebbe certamente possibile realizzare un sistema "appetibile" per il mercato senza un microcontrollore in grado di gestire le sequenze operative dei vari programmi di lavaggio, delle impostazioni dell'utente tramite tastiera/display e, per le applicazioni più moderne, la comunicazioni di dati di funzionamento direttamente attraverso la rete elettrica (v. **power-line communication**).
△

▽ *Esempio: Controllo dell'iniezione di carburante nei motori a scoppio*

Nei motori a scoppio la generazione della forza motrice è basata sulla compressione e combustione di una miscela composta di aria e carburante. Per ottenere una combustione ottimale, in relazione soprattutto alla composizione chimica dei gas di scarico, è necessario che le quantità di aria e di carburante, nella miscela che entra nei cilindri del motore, siano sempre in rapporto tra loro con un valore numerico che dipende dal tipo di carburante impiegato. Ad esempio, con la benzina tale rapporto (detto **rapporto stechiometrico**) deve essere pari a 14,7:1. Ovviamente, il sistema di alimentazione del motore deve essere in grado di fornire sempre una miscela aria/carburante che rispetti il rapporto stechiometrico, regolando le quantità dei due componenti in base alle condizioni di funzionamento del motore ed alle condizoni ambientali (temperatura, pressione atmosferice, ecc.). Il sistema di alimentazione tradizionalmente impiegato, il *carburatore*, fornisce una miscela la cui composizione dipende dal **volume** dell'aria aspirata ed dal **volume** del carburante dosato dagli spruzzatori interni al carburatore stesso. È evidente che tale sistema, pur con i livelli di sofisticatezza raggiunti dai recenti progetti meccanici, non è in grado di compensare con precisione le variazioni di pressione e temperatura, che inevitabilmente influenzano le quantità effettive di aria e combustibile contenute nello stesso volume.

I moderni sistemi di alimentazione (*Electronic Fuel Injection* o EFI) sono basati quindi su un

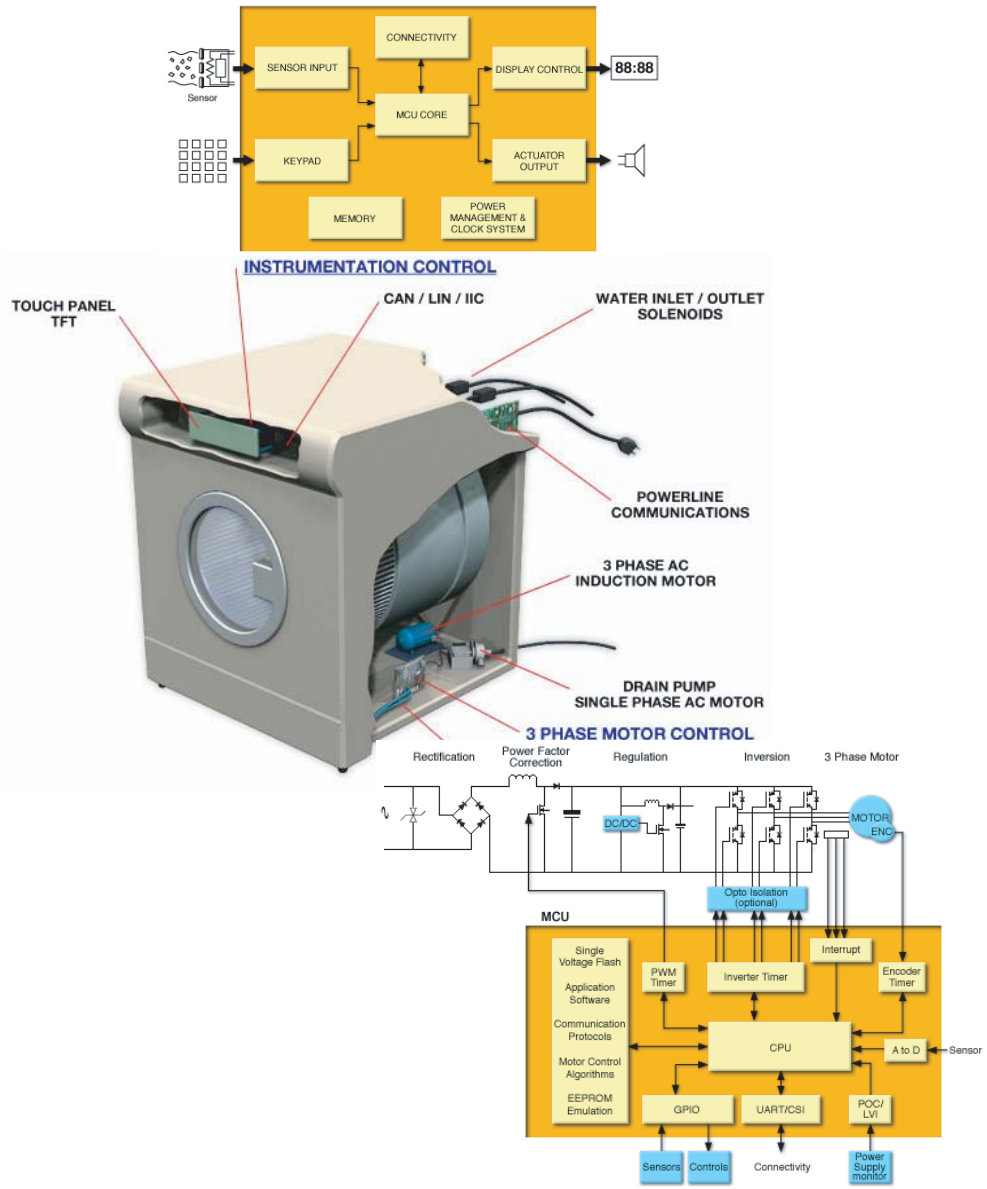


Figura B.2.4: Microcontrollore per uso domestico: controllo motore cestello e gestione programma di lavaggio in una lavatrice

microcontrollore (*Engine Control Unit* o ECU) in grado di regolare la miscela aria/carburante grazie ad opportuni sensori ed attuatori. Osservando la Figura B.2.5, è possibile analizzare la tipologia ed il ruolo di tali dispositivi:

- il carburante viene prelevato dal serbatoio (`Tank`) grazie alla pompa `Low Pressure Fuel Pump`, comandata direttamente dall'ECU;
- la differenza di pressione tra i condotti a valle e a monte dell'iniettore (`Injectors`), viene mantenuta costante grazie ad un regolatore di pressione (`High Pressure Fuel Pump`) e misurata tramite il sensore `Fuel High Pressure Sensor`; in questo modo, la quantità di carburante che entra nel cilindro dipende solo dal tempo di apertura dell'otturatore dell'iniettore (azionato da un elettromagnete `Pencil Coil` comandato dall'ECU);

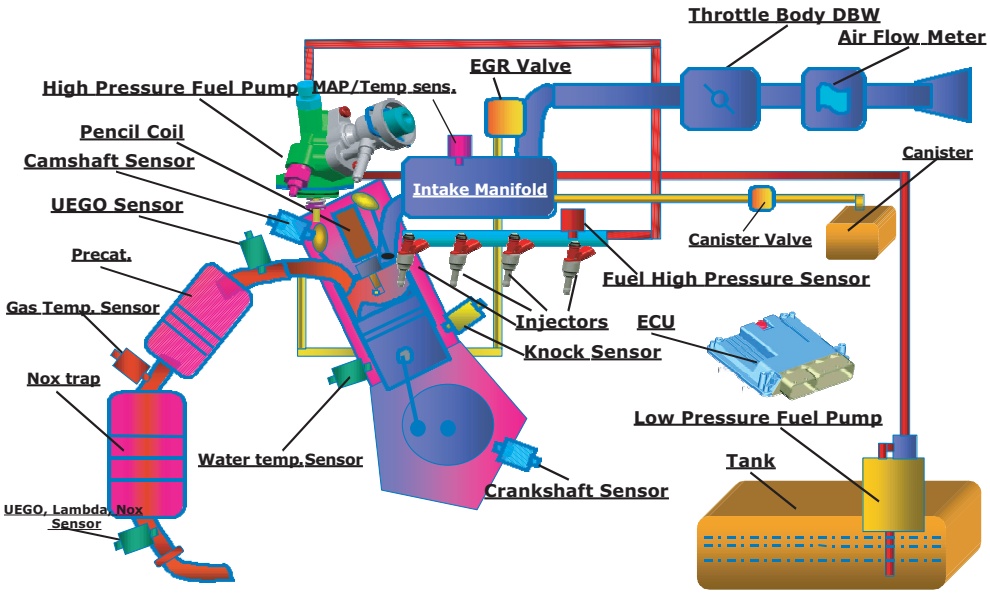


Figura B.2.5: Schema dei componenti di un sistema di iniezione carburante

- la quantità di aria che entra nel condotto di aspirazione (`Intake Manifold`), dipende da tre fattori:
  - la posizione di una valvola a farfalla, collegata all'acceleratore del guidatore, misurata dal sensore `Throttle Body DBW`, chiamato anche TPS (Throttle Position Sensor);
  - il flusso dell'aria, misurato dal sensore `Air Flow Meter`;
  - la temperatura e pressione dell'aria, misurata direttamente nel condotto di aspirazione dal sensore `MAP/Temp Sensor`, per il quale MAP indica Manifold Absolute Pressure;
- note le informazioni relative alla quantità di aria aspirata e regolata la pressione del carburante, la ECU può attivare gli iniettori per il tempo necessario durante la fase di aspirazione di ciascun cilindro, che viene rilevata grazie al sensore di posizione dell'albero motore (`Crankshaft Sensor`) ed e al sensore di identificazione del cilindro (`Camshaft Sensor`);
- l'effettiva qualità della combustione viene misurata nel condotto di scarico in diversi punti, grazie a sensori `UEGO` (Universal Exhaust Gas Oxygen), che tipicamente misurano la concentrazione di ossigeno (O_2) nei gas di scarico; grazie a questa informazione, infatti, è possibile determinare se la miscela aria/carburante iniettata nei cilindri era effettivamente prossima al rapporto stechiometrico (per tale motivo il sensore di O_2 è spesso chiamato sonda λ, essendo λ il simbolo tipicamente associato al rapporto stechiometrico).
- altre informazioni utili per la ECU sono la temperatura del liquido di raffreddamento (`Water Temp. Sensor`), dei gas di scarico (`Gas Temp. Sensor`) e l'eventuale presenza di vibrazioni anomale (`Knock Sensor`) indicative di una combustione eccessivamente anticipata rispetto all'accensione della candela da parte della ECU;
- la Figura B.2.5 descrive anche un sistema per il ricircolo di gas di scarico (`EGR`, Exhaust Gas Recirculation) ed un sistema di filtraggio (`Canister`) dei vapori di carburante presenti nel condotto di aspirazione, il cui controllo permette di migliorare la qualità delle emissioni allo scarico.

Il tempo di attivazione degli iniettori (*injection pulse*) e quello di attivazione della candela di accensione (*ignition pulse*), sono entrambi regolati dalla ECU grazie a delle tabelle predefinite (la

cosiddetta *mappatura* della ECU), che memorizzano i valori di durata degli impulsi di comando in funzione delle grandezze misurabili più significative (posizione valvola a farfalla, velocità di rotazione del motore, MAP, ecc.). I valori delle tabelle, messe a punto dal costruttore del veicolo o da officine specializzate, sono poi opportunamente modificati dal software della ECU prima dell'attuazione, secondo fattori correttivi che dipendono principalmente dal feedback proveniente dal sensore di O_2, ma anche da parametri ausiliari (es. la temperatura del liquido di raffreddamento).

Ovviamente, la capacità di elaborazione ed interfacciamento della ECU viene solitamente sfruttata anche per altri funzionalità ausiliarie, come la gestione degli strumenti sul cruscotto o la diagnostica generale della parte elettrica del veicolo, come evidenziato dallo schema funzionale di Figura B.2.6.

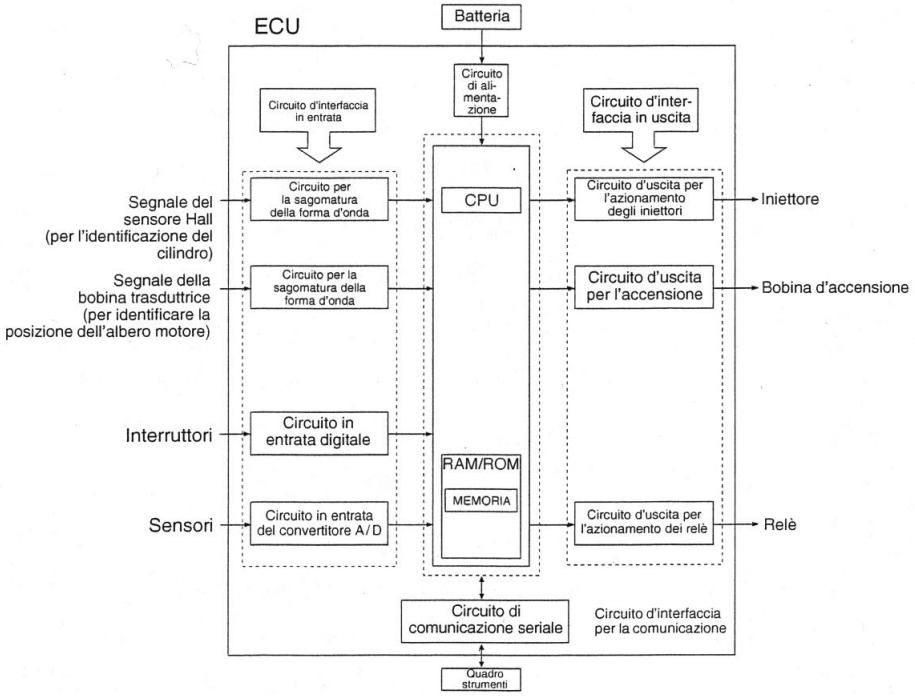


Figura B.2.6: Centralina per controllo iniezione carburante (Engine Control Unit)

△

B.2.2 Microcontrollori: architetture *storiche* e caratteristiche generali

Il primo dispositivo che fu effettivamente denominato *microcontrollore* dal proprio produttore fu l'**Intel 8051**, commercializzato nel 1980 e sviluppato sulla base del "core" della CPU ad 8 bit **Intel 8080**, integrando dei banchi di memoria RAM e ROM, dei dispostivi di conteggio eventi (Timer/Counter) ed un gestore di porta seriale, cioè l'insieme minimo di periferiche per poter realizzare operazioni di controllo logico (sensori digitali, attuatori "on/off").

Sebbene sia il capostipite delle successive generazioni di processori per il controllo, il "nome" 8051 è ancora oggi presente sul mercato, ed i dispositivi basati su tale architettura sono utilizzati correntemente in una percentuale consistente delle applicazioni di controllo custom. In effetti, i microcontrollori basati sul nucleo ("core") originario dell'8051 sono oggi prodotti da un gran numero di costruttori (ad inizio 2022 se ne contano più di 70[4], dalla ABOV Semiconductor alla Zylogic Semiconductor, passando per Infineon e Texas Instruments), che hanno acquistato dalla Intel il

[4]Si veda l'elenco alla pagina http://www.keil.com/dd/chips/all/8051.htm.

progetto del *core* e delle sue successive estensioni. Va anche osservato che per l'8051, essendo così noto e diffuso da molto tempo, esistono anche molti compilatori gratuiti o addirittura open-source[5], cosa non sempre vera per microcontrollori più moderni (i cui strumenti di sviluppo professionali sono spesso proprietari e molto costosi).

Ovviamente, il successo di tale processore e dei suoi successori deriva dalla stretta relazione con la classica architettura **Intel x86/x64**, ancora oggi dominante sul mercato dei processori per PC e pertanto quella maggiormente conosciuta dai progettisti hardware di sistemi di elaborazione.

I dati tecnici delle caratteristiche basilari dell'8051 del 1980, il cui schema a blocchi semplificato è mostrato in Figura B.2.7, si possono riassumere come segue:

- CPU ad 8 bit.
- Set istruzioni specializzato per la logica booleana ad 1 bit.
- 64 Kb di massimo spazio di memoria indirizzabile.
- 4 Kb di ROM[6] On-Chip (*Program o Code Memory*).
- 128 byte (esatto, byte! 256 byte nella versione 8052) di RAM On-Chip (*Data Memory*), dei quali 128 bit sono indirizzabili individualmente grazie alle istruzioni booleane.
- 4 Porte di I/O ad 8 bit, indirizzabili anche singolarmente (⇒ fino a 32 I/O digitali).
- 2 contatori di eventi (Timers/Counters) a 16 bit.
- una porta seriale programmabile (Full-Duplex UART, cioè Universal Asynchronous Receiver/-Transceiver).
- Interrupt Controller per gestire 5 diversi eventi.
- Oscillatore On-Chip (collegabile ad un quarzo risuonatore esterno ⇒ Clock 12 ÷ 20 MHz).

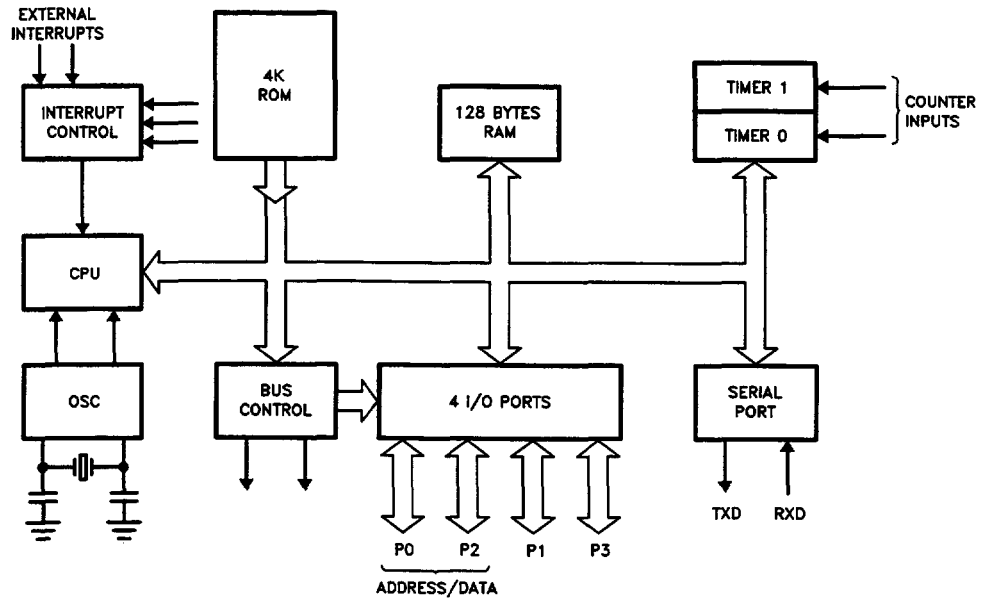


Figura B.2.7: Diagramma a blocchi schematizzato dell'architettura interna 8051 (fonte Intel)

[5] Si veda ad esempio il progetto Small Device C Compiler (SDCC): http://sdcc.sourceforge.net/.
[6] Con l'evoluzione della tecnologia costruttiva, poi realizzata con banchi riscrivibili EEPROM o Flash.

B.2.2.1 Architettura Intel 8051

L'architettura 8051 è quella "classica" dei primi processori Intel, con un bus interno istruzioni/dati ad 8 bit, mentre gli indirizzi a 16 bit permettono di ottenere la dimensione dello spazio di indirizzamento, cioè la quantità massima di memoria accessibile dal dispositivo, di 64 Kb, sia per la *Data Memory* che per la *Program Memory*. Lo spazio di indirizzamento di istruzioni e dati è separato, vale a dire che un byte in memoria dati può avere lo stesso indirizzo di una istruzione di programma. Questo ovviamente non crea problemi, in quanto le due parti di memoria sono anche fisicamente distinte. Occorre però ricordare che il bus di sistema è comunque unico, pertanto l'architettura fisica dell'8051 è di fatto una architettura Von Neumann, anche se dal punto di vista *logico* (i.e. spazio di indirizzamento) recepisce alcune caratteristiche del tipo Harvard.

Come detto, il chip del processore 8051 (versione "base") è dotato internamente di 4 Kb di memoria ROM, per il codice di programma, e di soli 128 byte di RAM. Tuttavia, il dispositivo è anche in grado di accedere a memoria esterne, come detto fino ad un massimo di 64 Kb, sfruttando (e pertanto "sacrificando") i 16 pin delle porte P0 e P2 (v. Figura B.2.7) come interfaccia parallela con indirizzi e dati multiplexati.

Le istruzioni memorizzate nella ROM, sia essa interna od esterna, agli indirizzi più bassi sono di fondamentale importanza: la prima istruzione eseguita all'avvio del dispositivo è sempre quella con indirizzo 0000H[7], mentre le istruzioni con indirizzi da 0003H a 0033H sono riservate alla programmazione delle *routine di gestione degli interrupt* (**Interrupt Service Routine o ISR**), che costituiscono la **Interrupt Vector Table (IVT)** (v. Figura B.2.8 e successiva sezione B.2.2.4). Si noti che la dimensione della zona di Program Memory riservata a ciascun evento di interrupt è di soli 8 byte: se questi non sono sufficienti per contenere la routine di risposta, sarà necessario programmare un salto ad una qualunque altra zona della memoria ROM di programma.

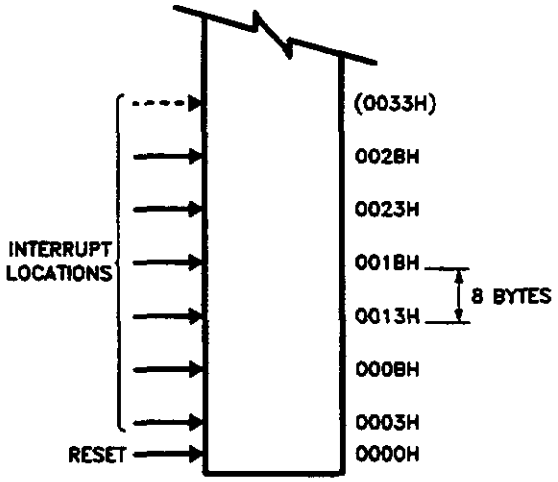


Figura B.2.8: Locazione in Program Memory delle routine di gestione interrupt nell'8051 (fonte Intel)

Per quanto riguarda la memoria dati (RAM), invece, il suo ruolo è molto differente a seconda che si consideri quella interna o quella esterna al chip. La RAM interna, infatti, è divisa in tre blocchi: la **Lower RAM** (i 128 byte inferiori), la **Upper RAM** (i 128 byte superiori, dove presenti, es. 8052), e la zona riservata per i **registri di funzionamento speciali** (**Special Function Registers** o **SFR**, v. sezione B.2.2.3). La Upper RAM e la zona degli SFR sono allocate nello stesso spazio di indirizzamento, ma sono fisicamente separate ed il processore ne distingue l'accesso in base alla modalità di indirizzamento

Il ciclo macchina dell'8051, cioè la sequenza delle fasi di prelievo (**fetch**) dell'istruzione, prelievo dei dati ed esecuzione vera e propria (**execute**), è costituito da 6 stati, per ognuno dei quali sono richiesti due periodi del clock di sistema, come mostrato dalla Figura B.2.10. Pertanto ogni ciclo

[7] Con xxxxH si intende la notazione esadecimale.

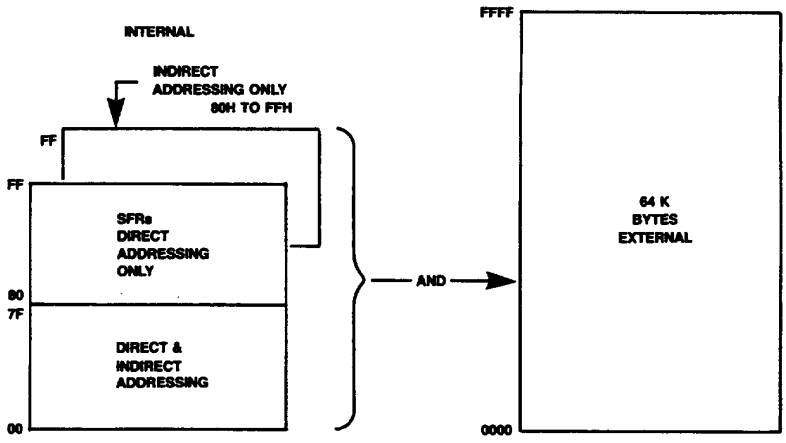


Figura B.2.9: Organizzazione completa della RAM interna dell'8051 (fonte Intel)

macchina, che corrisponde all'esecuzione di una istruzione "base", richiede 12 cicli di clock, vale a dire, con un clock (tipico) a 12 MHz, 1 μs. Poichè solitamente si indica la velocità di calcolo di un processore in milioni di istruzioni eseguite nell'unità di tempo (Mega-Instructions Per Second o MIPS), è possibile affermare che un 8051 a 12 MHz ha una velocità di calcolo di 1 MIPS.

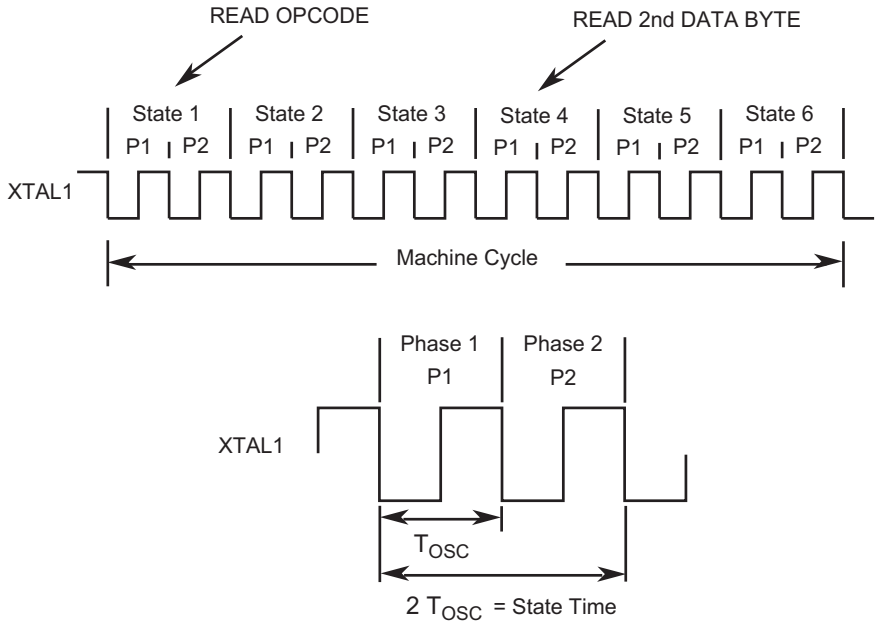


Figura B.2.10: Sequenza temporale del ciclo macchina dell'8051

Le istruzioni base dell'8051 sono composte sempre da un operatore (**opcode**) e da uno o due operandi, dei quali il primo costituisce implicitamente anche l'elemento che conterrà il risultato dell'operazione. Ad esempio, l'istruzione aritmetica di somma è indicata nel linguaggio **Assembly** per l'8051 come segue:

```
1 ADD A,#55
```

e corrisponde ad incrementare il valore contenuto nel registro **A** (anche indicato con **ACC**, da *accumulator*) del valore (costante) di 55.

B.2.2.2 Architetture *moderne*: esempio Microchip PIC

Con il progredire della tecnologia a semiconduttore e con le sempre crescenti esigenze dell'automazione, sia nell'industria (per il controllo sofisticato di azionamenti elettrici), sia in campo automobilistico e domestico, è cresciuta di pari passo la dotazione di periferiche integrate dei moderni microcontrollori, la capacità di elaborazione dati a risoluzione elevata (16-32 bit) e le frequenze di clock. Il mercato attuale offre processori orientati al controllo dotati di:

- Convertitori A/D e D/A, con estensione a più ingressi tramite multiplexer.
- Diversi Timer/Counter, tra i quali generalmente un timer dedicato ad operazioni di **Watchdog**[8].
- Diverse uscite per la generazione di impulsi modulati in ampiezza (**Pulse Width Modulation** o **PWM**), anche sincronizzabili tra loro (funzionalità tipicamente necessaria per pilotare i circuiti di potenza "switching" per azionamenti elettrici e convertitori in genere, DC/AC o DC/DC).
- porte di comunicazione per diversi protocolli standardizzati (es. I^2C, CAN).
- Controllori autonomi per la gestione di display LCD.

I maggiori produttori (o *vendor*) di microcontrollori moderni sono quelli menzionati nella Figura B.2.11, che fa riferimento a recenti statistiche sulla suddivisione percentuale del mercato. In cima alla lista si trovano NXP/Freescale, Renesas (azienda giapponese nata come fusione di Hitachi e NEC) e Microchip/Atmel (concorrenti storiche, ora unite sotto la proprietà della prima).

Leading MCU Suppliers ($M)

2016 Rank	Company	2015	2016	% Change	% Marketshare
1	NXP*	1,350	2,914	116%	19%
2	Renesas	2,560	2,458	-4%	16%
3	Microchip**	1,355	2,027	50%	14%
4	Samsung	2,170	1,866	-14%	12%
5	ST	1,514	1,573	4%	10%
6	Infineon	1,060	1,106	4%	7%
7	Texas Instruments	820	835	2%	6%
8	Cypress***	540	622	15%	4%

*Acquired Freescale in December 2015.

**Purchased Atmel in April 2016.

***Includes full year of sales from Spansion acquisition in March 2015.

Source: IC Insights, company reports

Figura B.2.11: Classifica di mercato dei maggiori produttori di microcontrollori

Un'altra statistica altrettanto interessante è quella relativa alla suddivisione del mercato fra microcontrollori a 8, 16 o 32 bit. Come si può notare dalla Figura B.2.12, il mercato dei dispositivi a 8 bit è ancora in espansione, sebbene il suo peso in percentuale stia calando a favore di quelli a 16 e 32 bit[9]. Ciò dimostra anche che i processori ad 8 bit, sebbene abbiano le architetture più semplici

[8] Il Watchdog ("cane da guardia") è un timer programmabile, il cui overflow provoca il **reset** automatico del processore. Il software programmato dall'utente deve perciò reinizializzare il timer prima dell'overflow. Se ciò non viene fatto, significa che il programma ha superato il proprio vincolo di *real-time*, condizione indice di possibili errori software. Il Watchdog costituisce quindi un dispositivo di sicurezza, che ovviamente può anche non essere abilitato se l'applicazione non lo richiede.

[9] Dovuta soprattutto alla massiccia introduzione di unità con architettura **ARM**

e le capacità di calcolo inferiori, avranno anche in futuro una notevole importanza nelle applicazioni di controllo.

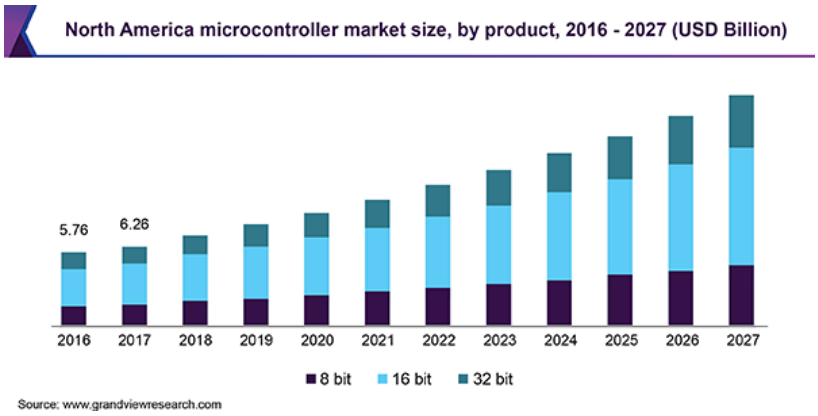


Figura B.2.12: Previsioni di mercato (Nord America) dei microcontrollori a 8, 16 e 32 bit

Considerando appunto tale classificazione in base alla dimensione delle unità di dati elaborati, si possono così riassumere le offerte commerciali più interessanti:

1. **Microcontrollori a 8 bit**:
   - **Atmel** (ora parte del gruppo Microchip): la sua architettura specifica denominata **AVR** è alla base del *best-seller* tra le schede per hobbistica: **Arduino**.
   - **Microchip:** i suoi microcontrollori ad 8 bit più noti sono quelle della famiglia **PIC**.
   - **Renesas:** oltre alla famiglia H8 "storica" (derivata da Hitachi), produce le nuove famiglie RL78 e 78K.
   - **STMicroelectronics:** la sua architetture STM8 è fortemente orientata alle applicazioni automotive.
2. **Microcontrollori a 16 bit**:
   - **Microchip:** l'architettura PIC è stata sviluppata e potenziata per creare non solo i microcontrollori **PIC24**, ma anche veri e propri **Digital Signal Controller (DSC)**, delle famiglie **dsPIC30** e **dsPIC33**.
   - **Texas Instruments:** uno dei principali produttori di **Digital Signal Processor (DSP)**, fornisce microcontrollori orientati alle applicazioni *low-power*, gli **MSP430**.
3. **Microcontrollori a 32 bit**: in questa fascia, gli ultimi anni hanno visto la prepotente irruzione di processori con architettura **ARM Cortex-Mx**. La compagnia che ha sviluppato tale architettura non produce effettivamente semiconduttori, ma vende sotto licenza il progetto *core* della CPU alla maggior parte dei produttori citati anche in precedenza:
   - **Atmel:** oltre ai dispositivi basati sull'estensione dell'architettura AVR a 32 bit (**AVR32**), la propria famiglia **SAM3** ha un core ARM Cortex-M3 ed è alla base della nuova generazione del già citato *best-seller* di controllori per "creativi" dell'elettronica: **Arduino Due**.
   - **Microchip:** anzichè acquistare la licenza ARM, ha investito sullo sviluppo di una propria architettura per i **PIC32**.
   - **Texas Instruments:** sebbene l'azienda li definisca anche microcontrollori, i suoi dispositivi della famiglia **C2000** sono di fatto dei DSP/DSC, mentre la nuova famiglia **Tiva** è basata sulla recentissima architettura ARM Cortex-M4.

- **STMicroelectronics:** anch'essa propone una variante su base ARM Cortex-M3/M4, chiamata **STM32**.

Verrà nel seguito descritta brevemente l'architettura Microchip PIC ad 8 bit, di interesse per le notevoli differenze rispetto a quella più tradizionale dell'Intel 8051, oltre che per il suo ulteriore sviluppo nella famiglia dsPIC33, oggetto delle esercitazioni per questo corso.

I microcontrollori PIC sono caratterizzati principalmente dall'architettura interamente **Harvard**, vale a dire che oltre ad avere spazi distinti per Code Memory e Data Memory hanno anche due bus indipendenti per l'accesso a tali memorie, e dal ridotto set di istruzioni base (**Reduced Instruction Set Computer, RISC**), in contrasto con l'8051 che ha invece un unico bus di sistema ed un compesso set di istruzioni (**Complex Instruction Set Computer, CISC**).

L'architettura Harvard dei PIC, visualizzata in Figura B.2.13 in confronto con quella 8051, permette, tra le altre cose, di distinguere la dimensione delle singole locazioni di memoria per istruzioni e dati. Ad esempio, le *word* di codice possono avere lunghezza di 12, 14 o 16 bit, mentre le locazioni di memoria sono sempre ad 8 bit. Le dimensione delle istruzioni base definisce anche la classificazione delle varie famiglie di PIC, denominate da Microchip come segue:

- **Baseline**, nomi identificativi dei dispositivi con prefisso PIC10 e PIC12, istruzioni a 12 bit[10];
- **Mid-range**, nomi identificativi dei dispositivi con prefisso PIC16, istruzioni a 14 bit;
- **High-end**, nomi identificativi dei dispositivi con prefisso PIC18, istruzioni a 16 bit;

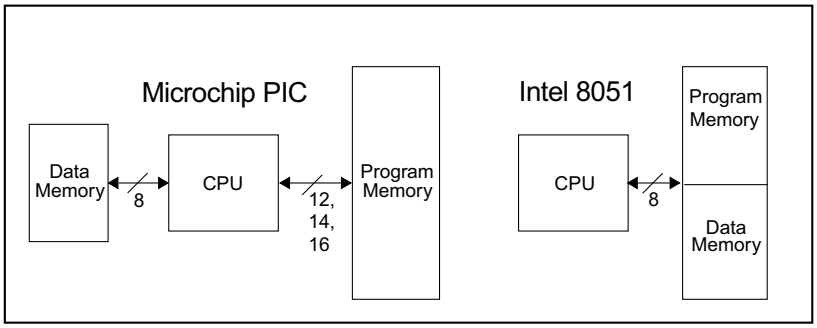


Figura B.2.13: Confronto fra architettura PIC e 8051

In dettaglio, si può considerare come esempio lo schema a blocchi completo di una delle famiglie PIC Mid-Range, riportato in Figura B.2.14. Si noti la presenza di una grande quantità e varietà di periferiche (ADC, Timers, driver per LCD, ecc.) e di funzionalità per operare in modalità low-power.

Un altro fattore di distinzione fra l'architettura PIC e quella 8051 è la gestione (più efficiente nel caso dei PIC) del ciclo macchina. I PIC sono infatti in grado di completare una istruzione in 4 periodi di clock (anzichè i 12 dell'8051), grazie anche alla organizzazione in **pipeline** delle operazioni di **fetch** ed **execute** (tipica delle architetture di CPU moderne), come mostrato in Figura B.2.15.

Da un altro lato, nella maggior parte dei PIC (salvo pochi dispositivi PIC18) non esiste un meccanismo di indirizzamento ed accesso a memorie esterne che sia direttamente gestito via hardware, il che ne limita fortemente le possibilità di espansione. L'accurata selezione del microcontrollore PIC più idoneo per una certa applicazione, in funzione delle previsioni su lunghezza del codice e quantità di dati necessari, diventa quindi una procedura delicata ed importante.

L'architettura PIC è come detto di tipo RISC, cioè a ridotto set di istruzioni: da un minimo di 35 istruzioni ad un massimo di 70 per i dispositivi *High-end*. Una tipica istruzione Assembly per PIC include un codice operatore (**opcode**) di lunghezza variabile ed un operando, tipicamente indicato come **file register**, se associato all'indirizzo di una locazione di memoria dati. In molti casi, il secondo operando implicito dell'istruzione è il registro accumulatore **W**, secondo la denominazione Microchip,

[10]Si noti che i prefissi dei nomi di PIC includono un numero a due cifre, che però NON coincide in genere con la dimensione delle istruzioni.

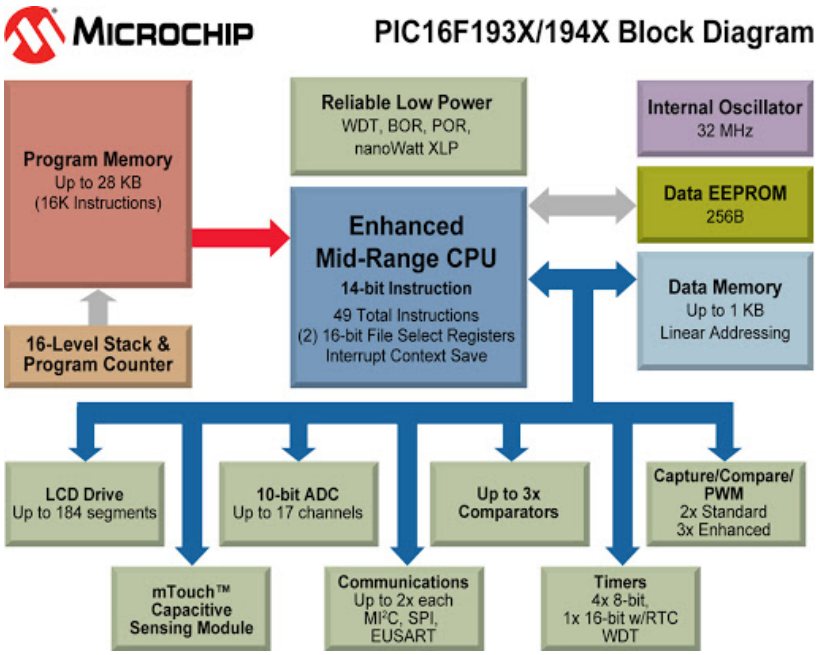


Figura B.2.14: Schema a blocchi di una delle versioni di PIC16 (*core* 8 bit con istruzioni di 14 bit)

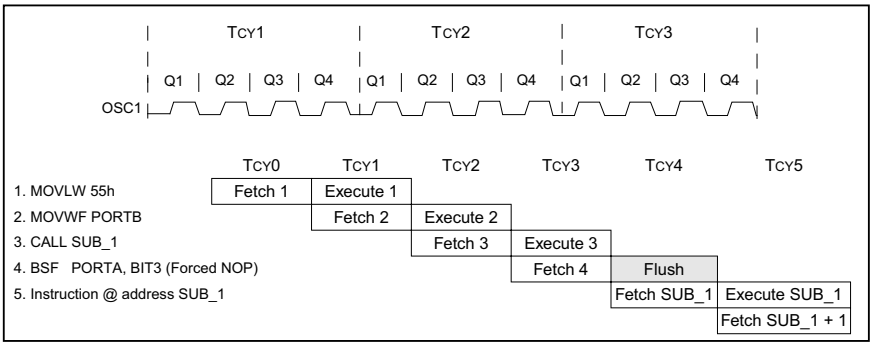


Figura B.2.15: Organizzazione del ciclo macchina nell'architettura PIC

cioè il registro nel quale viene sempre memorizzato il risultato di ogni operazione logico-aritmetica. Ad esempio, l'istruzione:

```
1 ADDWF 0x0C,0
```

somma all'accumulatore W il valore memorizzato in RAM all'indirizzo 0CH (esadecimale) e pone il risultato in W. Si noti che lo 0 che chiude l'istruzione non indica un valore numerico di un operando, ma il flag che specifica la destinazione del risultato. Se impostato a 1, come segue:

```
1 ADDWF 0x0C,1
```

l'istruzione pone il risultato nella locazione di memoria all'indirizzo 0CH, anzichè in W.

B.2.2.3 Periferiche integrate e Special Function Registers (SFR)

La caratteristica fondamentale dei microcontrollori, dall'8051 a tutti i suoi successori indipendentemente dall'architetture di base, è rappresentata dall'integrazione di periferiche utili per il controllo in tempo reale, come temporizzatori, contatori di eventi, convertitori A/D, generatori di impulsi PWM

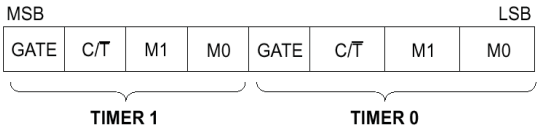
e così via. Queste periferiche sono gestite in base alla necessità dell'applicazione tramite i cosiddetti registri di funzionamento speciali o **Special Function Registers (SFR)**. Quest'ultimo termine, introdotto nei manuali di programmazione della Intel per l'8051, è ad oggi di uso comune per tutti i microcontrollori di qualunque *vendor*. Occorre anche notare che nella maggior parte dei casi gli SFR sono in realtà locazioni riservate nella memoria RAM, come appunto osservato in precedenza in relazione alla Figura B.2.9 per l'8051.

Per la configurazione di una delle periferiche integrate in un microcontrollore può essere necessario modificare il valore di uno o più SFR, il cui contenuto va specificato generalmente bit per bit. In altre parole, negli SFR devono essere scritti via software opportuni valori binari che ne identificano le modalità di funzionamento (es. contatore di impulsi esterni piuttosto che timer) ed i parametri numerici (es. valore limite di conteggio per un timer, velocità in bit per secondo della comunicazione per una interfaccia seriale, ecc.). Inoltre, poichè molti tipi di periferica operano parallelamente all'esecuzione delle istruzioni nella CPU, gli SFR permettono anche allo stesso programma in esecuzione sulla CPU di accedere ai risultati elaborati dalle periferiche (es. valore in uscita da un convertitore A/D, byte ricevuti tramite una interfaccia di comunicazione seriale, ecc.).

A titolo puramente esemplificativo, si considerano nel seguito i **Timer/Counter** della versione base dell'8051 (due, denominati *Timer0* e *Timer1*). La configurazione di queste due periferiche, effettuata attraverso gli SFR **TMOD** (Timer MODe) e **TCON** (Timer CONtrol), prevede anzitutto la programmazione della modalità di funzionamento, che viene distinta fra **"timer"**, nella quale l'incremento del valore di conteggio avviene automaticamente (indipendentemente dal software o da eventi esterni) ad ogni **ciclo macchina** e **"counter"**, nella quale l'incremento avviene al **fronte di discesa di un pin** di ingresso (i.e. `P3.4` per il Timer0 e `P3.5` per il Timer1). L'abilitazione/disabilitazione al conteggio di Timer0 e Timer1 avviene tramite due bit separati all'interno del registro TCON. Inoltre, la gestione del valore di conteggio, che viene mantenuto in formato a 16 bit negli SFR **TH0** (Timer0 High byte), **TL0** (Timer0 Low byte), **TH1** (Timer1 High byte) e **TL1** (Timer1 Low byte), e della condizione di azzeramento e riavvio (i.e. **overflow** o **roll-over**: raggiunto il valore massimo rappresentabile, es. FFH a 8 bit, l'incremento successivo riporta il valore di conteggio a 0) può essere configurata per funzionare in una delle seguenti modalità:

- **Mode 0:** il valore di conteggio viene mantanuto in un formato a 13 bit, sfruttando i primi 5 del registro TL0(1) e gli 8 del registro TH0(1), pertanto il valore massimo di conteggio è 2^{13}; raggiunto tale valore di roll-over, viene settato automaticamente un bit di overflow contenuto nel registro **TCON** e, soprattutto, viene generato un interrupt.
- **Mode 1:** analogo, ma il valore di conteggio è memorizzato usando a pieno TH0(1) e TL0(1), pertanto il valore di roll-over è 2^{16}.
- **Mode 2:** (o *auto-reload mode*) il valore di conteggio è memorizzato solo in TL0(1), per cui il massimo è 2^8, ed al momento del roll-over il registro viene re-inizializzato al valore memorizzato nel registro TH0(1), che non viene mai modificato se non via software.
- **Mode 3:** questa modalità può essere impostata solamente per Timer0, e permette di utilizzare TL0 e TH0 come due contatori separati; se il Timer0 è in questa modalità, i due bit di TCON che abilitano o disabilitano al conteggio sono entrambi riferiti ad esso (conteggio in TL0 e conteggio in TH0), mentre Timer1 rimane pertanto sempre disabilitato.

La Figura B.2.16 mostra un estratto della documentazione originale della Intel che descrive il ruolo dei bit nel registro TMOD. Ad esempio, assegnando a TMOD il valore esadecimale 31H (in binario 0011 0001), si pone il Timer1 in Mode 3 (M1/M0 entrambi a 1 per il Timer 1: disabilitato) e il Timer0 in Mode 1 (M1/M0 = 01 per il Timer 0: funzionamento a 16-bit).



	GATE	Gating control when set. Timer/Counter "x" is enabled only while "INTx" pin is high and "TRx" control pin is set. when cleared Timer "x" is enabled whenever "TRx" control bit is set.
	C/T	Timer or Counter Selector cleared for Timer operation (input from in=ternal system clock.) Set for Counter operation (input from "Tx" input pin).
M1	**M0**	**OPERATING**
0	0	8048 Timer "TLx" serves as 5-bit prescaler.
0	1	16-bit Timer/Counter "THx" and "TLx" are cascaded; there is no prescaler.
1	0	8-bit auto-reload Timer/Counter "THx" holds a value which is to be reloaded into "TLx" each time it overflows.
1	1	(Timer 0) TL0 is an 8-bit Timer/Counter controlled by the standard Timer 0 control bits. TH0 is an 8-bit timer only controlled by Timer 1 control bits.
1	1	(Timer 1) Timer/Counter 1 stopped.

Figura B.2.16: Dettaglio della documentazione 8051 per SFR TMOD

Come esempio di periferica di importanza fondamentale nei sistemi embedded per il controllo, ma non presente nel nucleo originario dell'Intel 8051, va certamente citato il convertitore A/D. Poichè nei microcontrollori di tipo PIC tale dispositivo è quasi sempre integrato, si presenta per riferimento lo schema di Figura B.2.17, che mostra il convertitore A/D di un PIC Mid-Range. Si noti che i pin utilizzabili come ingressi analogici sono 8 e sono connessi all'ADC tramite un multiplexer. Le tensioni di riferimento massima e minima, $\mathbf{V_{REF+}}$ e $\mathbf{V_{REF-}}$ (i.e. l'intervallo di tensioni convertite), dell'ADC possono essere connesse all'alimentazione del chip (i.e. $\mathbf{V_{DD}}$ e $\mathbf{V_{SS}}$) o a due pin di ingresso, modalità che può essere utile per scalare i segnali acquisiti rispetto a valori di riferimento forniti da generatore di tensione esterni.

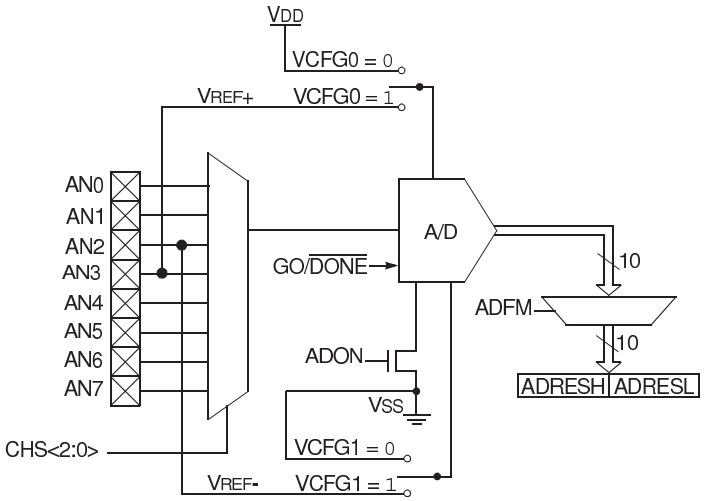


Figura B.2.17: Il sistema di acquisizione per segnali analogici di un microcontrollore PIC

La configurazione degli ingressi analogici di un PIC avviene tramite i registri SFR **ADCON0**, **ADCON1** e **ANSEL**. Il primo di questi SFR (vedi Figura B.2.18) contiene i 3 bit **CHS2:CHS0** per l'indirizzamento del multiplexer (i.e. $2^3 = 8$ input analogici), il bit **ADON** che abilita il convertitore e il bit **GO/$\overline{\text{DONE}}$** che, se forzato a 1 via software, determina l'avvio della conversione vera e propria (il convertitore A/D è ad approssimazioni successive) e al contempo segnala il termine della conversione da parte dell'ADC (che lo resetta via hardware). Il termine della conversione può anche essere causa di interrupt per il PIC, se opportunamente abilitato. Il registro **ADCON1**, qui non

descritto in dettaglio, permette di impostare la frequenza di clock del ciclo ad approssimazioni successive del convertitore e tramite i bit del registro **ANSEL** si definisce l'utilizzo dei pin associati al multiplexer come ingressi analogici o come I/O digitali. Il risultato della conversione, con risoluzione a 10 bit, viene automaticamente memorizzato nei due SFR ad 8 bit **ADRESH** ed **ADRESL**, allineando i 10 bit risultanti a destra o a sinistra a seconda dell'impostazione del bit **ADFM** (nell'SFR ADCON0).

REGISTER 12-2: ADCON0 – A/D CONTROL REGISTER (ADDRESS: 1Fh)

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
ADFM	VCFG1	VCFG0	CHS2	CHS1	CHS0	GO/$\overline{\text{DONE}}$	ADON
bit 7							bit 0

bit 7 **ADFM:** A/D Result Formed Select bit
1 = Right justified
0 = Left justified

bit 6 **VCFG1:** Voltage Reference bit
1 = V_{REF-} pin
0 = V_{SS}

bit 5 **VCFG0:** Voltage Reference bit
1 = V_{REF+} pin
0 = V_{DD}

bit 4-2 **CHS<2:0>:** Analog Channel Select bits
000 = Channel 00 (AN0)
001 = Channel 01 (AN1)
010 = Channel 02 (AN2)
011 = Channel 03 (AN3)
100 = Channel 04 (AN4)
101 = Channel 05 (AN5)
110 = Channel 06 (AN6)
111 = Channel 07 (AN7)

bit 1 **GO/$\overline{\text{DONE}}$:** A/D Conversion Status bit
1 = A/D conversion cycle in progress. Setting this bit starts an A/D conversion cycle.
This bit is automatically cleared by hardware when the A/D conversion has completed.
0 = A/D conversion completed/not in progress

bit 0 **ADON:** A/D Conversion Status bit
1 = A/D converter module is operating
0 = A/D converter is shut off and consumes no operating current

Figura B.2.18: Estratto del datasheet di un PIC16 con descrizione dell'SFR ADCON0

Infine, si riporta come ultimo aspetto della gestione di periferiche ed SFR, la logica di funzionamento dei pin di I/O di un PIC (valido come esempio generale di microcontrollore), descritta in dettaglio dallo schema di Figura B.2.19.

Il compito di tale circuito, interno al microcontrollore, è quello di permettere anzitutto l'impostazione della modalità **Input** piuttosto che **Output** del pin interessato. Tale operazione è effettuata tramite il dispositivo di campionamento digitale (*latch*) indicato nel circuito come **TRIS Latch**. Se a tale latch è assegnato un valore logico vero, si ottiene l'effetto di **disabilitare** entrambi i transistor in alto a destra (di tipo MOSFET, uno a canale P e uno a canale N). In tal modo, lo stato elettrico del pin risulta *flottante* e può essere forzato da un circuito esterno. In altre parole, il pin si configura come un *Input*. Si noti poi che il pin è collegato ai generici *Peripheral Modules* (es. il convertitore A/D, la UART, ecc.) tramite la linea in basso nella figura e che lo stato logico del pin può essere acquisito e trasferito al **Data bus** (in alto a sinistra) tramite il campionatore digitale collegato alla linea **RD PORT** (in basso a sinistra). Infine, si noti che se il pin è impostato in *Analog input mode*, un opportuno circuito logico ne inibisce la connessione al campionatore digitale di lettura dello stato, il cui valore viene quindi forzato a falso.

Se il *TRIS Latch* è impostato a vero, invece, lo stato logico del pin può essere forzato dal microcontrollore, quindi il pin è un *Output*. In tal caso, il valore vero/falso del pin corrisponde a quello assegnato al **Data Latch** (in alto a sinistra nella figura) e, conseguentemente, all'**abilitazione** di uno dei due transistor MOSFET già citati.

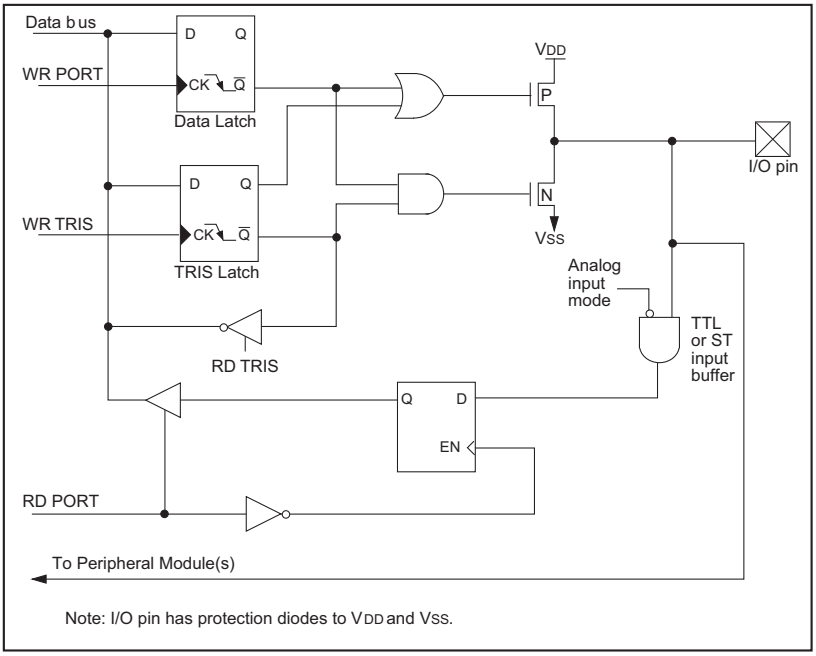


Figura B.2.19: Schema logico delle porte di I/O in un microcontrollore PIC

Questa logica di funzionamento viene generalmente detta **tri-state** (Input con stato logico vero, Input con stato logico falso, Output), definizione che spiega il termine *TRIS* associato nella figura al latch che imposta la direzione del pin. Nei PIC, gli SFR specifici collegati alla gestione delle porte sono quindi i seguenti:

1. **TRISx** (con **x** che indica la lettera associata alla porta, **A, B, C,** ecc.): registri di 8 bit collegati ai TRIS latch, impostano la direzione degli 8 pin associati alla porta. Se un certo bit di TRISx è a 1, il corrispondente pin della porta è utilizzabile come ingresso, se il bit è a 0, è una uscita[11].

2. **PORTx** (con **x** che indica la lettera associata alla porta, **A, B, C,** ecc.): registri di 8 bit che permettono di leggere lo stato dei pin associati alla porta, se configurati come input, o scrivere il valore vero/falso se configurati come output.

B.2.2.4 Gestione interrupt nei microcontrollori

Le periferiche integrate in un qualsiasi microcontrollore (i.e. 8051, PIC o generazioni ancora più moderne!) sono solitamente utilizzate in modo da alleggerire il compito di elaborazione della CPU, svolgendo operazioni specifiche (temporizzazione, conversione A/D, ricezione/invio dati tramite UART, ecc.) in modo indipendente dall'esecuzione del software. Per permettere alla CPU di sincronizzare l'elaborazione del software "principale" (i.e. la funzione `main()` di un programma in linguaggio C), ogni microcontrollore è dotato di un più o meno complesso **Interrupt Controller**. Gli *interrupt* sono eventi che forzano il processore a passare dall'esecuzione del programma principale all'esecuzione di opportune sequenze di istruzioni, chiamate **routine di gestione interrupt** (Interrupt Service Routine, ISR). Al termine dell'esecuzione di una ISR, l'esecuzione deve tornare esattamente allo stesso punto in cui si trovava il programma principale prima dell'interruzione, operazione che richiede ovviamente il **salvataggio/ripristino del contesto** di esecuzione (*context switching*), rispettivamente all'inizio e all'uscita della ISR. Il ruolo dell'Interrupt Controller è quello di intercettare gli eventi di interrupt (detti anche **Interrupt ReQuest, IRQ**), gestirne l'eventuale contemporaneità

[11] Un modo agevole per ricordare questa distinzione è visualizzare 1 come una I maiuscola (Input) e 0 come una O maiuscola (Output).

con logiche opportune (es. basate sulla priorità, di tipo First-In-First-Out o FIFO, ecc.) e preparare la CPU all'esecuzione dell'istruzione di partenza della ISR selezionata.

Gestione interrupt nell'8051

Nell'8051 le ISR sono memorizzate in locazioni prestabilite della memoria ROM (v. Figura B.2.8 sulla Interrupt Vector Table, IVT), che sono associate in modo rigido a ciascuna possibile sorgente di interrupt. In particolare, per l'8051 "classico" le cause di interrupt, ordinate come fissato nella IVT, sono le seguenti:

1. Fronte di salita/discesa sul pin esterno P3.2;
2. Overflow del Timer0;
3. Fronte di salita/discesa sul pin esterno P3.3;
4. Overflow del Timer1;
5. Completamento da parte della UART di ricezione/trasmissione di un carattere.

Ciascuna sorgente di interrupt può essere abilitata/disabilitata in modo indipendente dalle altre, attraverso bit opportuni dell'SFR **IE** (Interrupt Enable). Una volta fissato quali interrupt si desidera abilitare, è poi necessario che il bit **EA** (Enable All) nel registro IE sia impostato a 1, per avere l'effettiva abilitazione globale della gestione degli interrupt.

Una caratteristica molto importante nella gestione degli interrupt, soprattutto quando dipendono da molteplici cause, è la gestione delle loro priorità. Infatti, può capitare che:

1. si verifichi un nuovo evento di interrupt mentre il processore è già impegnato in una ISR;
2. due eventi di interrupt si verifichino contemporaneamente;

Nel primo caso, la scelta può essere quella di impedire sempre l'interruzione di una ISR e inibire di conseguenza l'annidamento tra le funzioni di ISR, oppure di permettere solo il passaggio da una ISR a bassa priorità ad una a più alta priorità (v. Figura B.2.20). Nell'8051 viene adottata quest'ultima politica, tramite la definizione di due livelli (**higher/lower priority level**) configurabili per ciascun evento di interrupt tramite i corrispondenti bit nell'SFR **IP** (Interrupt Priority). Per gestire il secondo caso, invece, l'8051 definisce un ordine di precedenza valido in caso di assoluta contemporaneità e fra eventi allo stesso livello di priorità, costituito dallo stesso ordine dell'elenco di eventi precedentemente riportato.

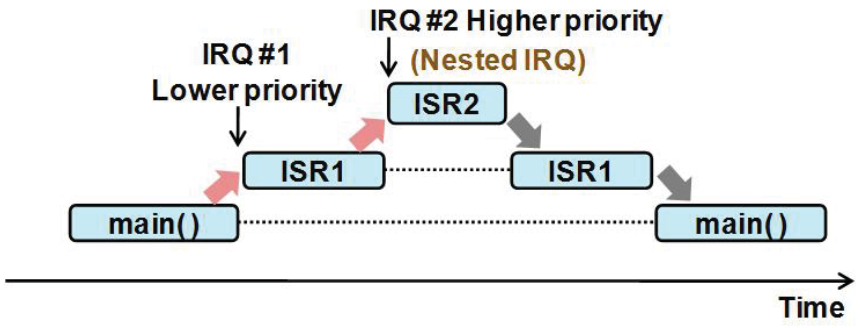


Figura B.2.20: Interruzione del `main()` da parte di due successive ISR, annidate tra loro per via del livello di priorità (possibile sequenza di esecuzione in un Intel 8051)

Infine, si noti che, in generale, una causa di interrupt per un microcontrollore non deve essere necessariamente gestita con una funzione allocata nella IVT. Questa soluzione infatti garantisce la massima prontezza di risposta, ma non è l'unica. Anche senza abilitare gli interrupt, è possibile

osservare il verificarsi di un evento, come l'overflow di un timer, controllando direttamente (con istruzioni software) un bit di *flag* specifico, solitamente memorizzato in un SFR per la gestione della periferica. Ad esempio, per il Timer0 dell'8051 l'SFR TCON contiene il flag TF0 (un bit) che viene settato via hardware quando il timer va in overflow. Se l'interrupt relativo è abilitato, il flag TF0 verrà **azzerato automaticamente una volta eseguita la relativa ISR**, pertanto la programmazione di quest'ultima **non deve includere istruzioni specifiche su tale flag**. Il seguente codice in linguaggio C, compatibile con il compilatore SDCC, mostra come potrebbe essere definita la ISR di gestione dell'overflow del Timer0:

```
void timer0_isr(void) interrupt TF0_VECTOR
{
  TL0 = 0;
  TH0 = 215;  // Impostazione TH0/TL0, in base al periodo desiderato
    LED = !LED; // Cambio stato LED (lampeggio)
}
```

In alternativa all'uso della ISR, lo stato del flag TF0 può essere valutato tramite lettura ciclica (i.e. **polling**) via software nel programma `main()`, **nel qual caso però dovrà anche essere azzerato via software**. Ad esempio, tale gestione a polling del flag TF0 corrisponde al seguente codice in linguaggio C:

```
...
while(TF0==0); // Attesa overflow Timer0 (ciclo vuoto)
               // ==> passaggio all'istruzione successiva
               // quando TF0 diventa 1
TF0 = 0;       // Azzeramento del flag di overflow Timer0
...
```

Si noti che la prima riga di codice costituisce una condizione **bloccante** per il programma principale. Al contrario, con una gestione ad interrupt il programma principale può continuare ad eseguire le proprie operazioni e sarà l'Interrupt Controller a forzarne l'interruzione per passare alla ISR.

Gestione interrupt nei PIC

La gestione degli interrupt nei PIC è meno articolata di quella dell'8051. Nei dispositivi *Baseline* e *Mid-range*, esiste una unica locazione di Program Memory riservata alla routine di risposta all'interrupt (locazione detta quindi **Interrupt Vector Address**, IVA, anzichè Table), indipendentemente dal numero di eventi che possono provocare l'interruzione del processore, mentre nei dispositivi *High-end* ne sono riservate due (una *High-priority* e una *Low-priority*). Ovviamente, nel primo caso non è prevista la possibilità di esecuzione annidata della routine di risposta agli interrupt.

Per tali motivi, è compito del progettista software programmare la (o le, nel caso *High-end*) routine in modo da distinguere via software le operazioni da eseguire in funzione dello stato dei vari flag di interrupt (overflow timer, ricezione seriale, ecc.). Come mostrato in Figura B.2.21, ciascuna sorgente di interrupt può essere abilitata o disabilitata in modo indipendente, tramite i bit con suffisso *IE* (*Interrupt Enable*, es. *ADCIE, TMR1IE*, ecc.) memorizzati in opportuni SFR, ed essere riconosciuta da uno specifico flag con suffisso *IF* (*Interrupt Flag*, es. *ADCIF, TMR1IF*, ecc.), ma tutti questi eventi vengono comunque valutati dall'Interrupt Controller con un blocco logico OR, per decidere l'interruzione della CPU. Il bit di SFR *GIE* (*Global Interrupt Enable*) abilita l'effettiva interruzione del processore ed il salto all'IVA, mentre gli eventi associati ad alcuni gruppi di periferiche possono essere disabilitati tramite il bit *PEIE* (*PEripheral Interrupt Enable*).
NOTA BENE: con tale logica di gestione degli interrupt, il PIC non può neanche resettare automaticamente i flag di interrupt all'esecuzione della ISR, come invece può fare l'Intel 8051, perciò il programmatore deve sempre inserire esplicitamente tale operazione nel codice della ISR stessa!

Un possibile schema generale per la programmazione della funzione di risposta agli interrupt in un PIC potrebbe essere quindi definito con il seguente listato in linguaggio C[12]:

[12] Il listato fornito è compatibile con il compilatore Microchip MPLAB XC8 per tutti i PIC a 8 bit. Si noti anche che utilizzando un compilatore per linguaggio C, è normalmente quest'ultimo ad occuparsi di inserire all'inizio della traduzione in linguaggio Assembly della funzione ISR le opportune istruzioni per il *context-switching*, che in caso di scrittura del codice direttamente in Assembly dovrebbero essere inserite dallo sviluppatore del software.

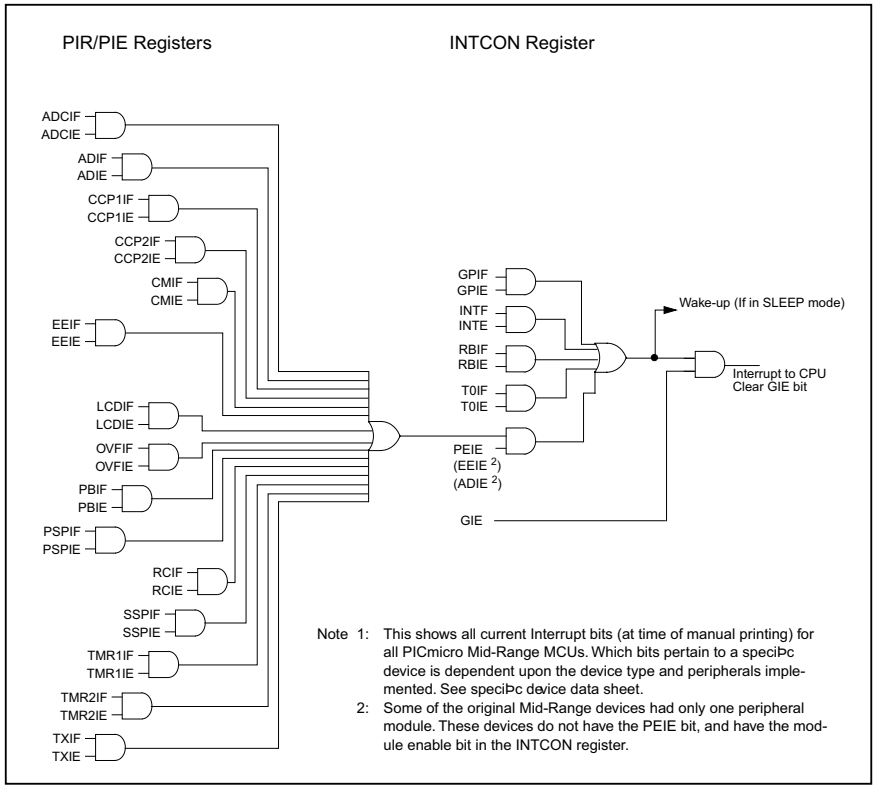


Figura B.2.21: Logica di gestione degli interrupt nei PIC

```
void interrupt ISR(void)
{
// xxxIF: Interrupt Flag, xxxIE: Interrupt Enabled
if(TMR1IE && TMR1IF) // Overflow Timer1
{
   TMR1IF = 0;  // reset del flag di interrupt: va eseguito via software
   TMR1L = 0;
   TMR1H = 215; // Riavvio per i prossimi 10ms
    ...
}
if(ADIE && ADIF) // Conversione ADC completata
{ ... }
if(RCIE && RCIF) // Ricezione carattere su porta seriale
{ ... }
}
```

B.2.2.5 Confronto riepilogativo tra Intel 8051 e Microchip PIC

Come detto, l'Intel 8051 è di interesse "storico" per capire la tecnologia dei microcontrollori e, seppure ancora presente sul mercato, è caratterizzato da alcune scelte architetturali obsolete, se confrontate con quelle di microcontrollori più moderni come i PIC. Da un altro lato, anche microcontrollori moderni come i PIC hanno limitazioni (es. la difficoltà di integrazione di memorie esterne o dispositivi simili ad accesso con data/address bit in parallelo). La dotazione di periferiche integrate nei PIC è però normalmente molto completa e, complessivamente, l'intera gamma di questi microcontrollori permette di selezionare il dispositivo idoneo per (quasi) qualunque applicazione.

Nella tabella B.2.1 sono riepilogate le caratteristiche che maggiormente differenziano tra loro 8051 e PIC, allo scopo di fornire una sintesi più facilmente memorizzabile.

	Intel 8051	Microchip PIC
Architettura	Von Neumann (Data/Program bus unico)	Harvard (Data e Program bus divisi)
Espandibilità	Possibile interfaccia con ROM/-RAM esterna	Non espandibile
Ciclo macchina	12 periodi di clock	4 periodi di clock
Set istruzioni	CISC	RISC
Gestione interrupt	IVT con 5 sorgenti e priorità	Un solo Interrupt Vector Address (due nei PIC18)
Reset interrupt flag	Automatica all'esecuzione della relativa ISR	Da programmare esplicitamente nella ISR

Tabella B.2.1: Sintesi di confronto 8051 - PIC

NOTA BENE: il confronto 8051 - PIC è qui proposto come esempio rappresentativo della variabilità di soluzioni architetturali e funzionali nella gestione di istruzioni, dati, periferiche e interrupt nel mondo dei microcontrollori. L'insegnamento che lo studente ne deve trarre è soprattutto la consapevolezza di dover studiare adeguatamente tali dettagli sull'architettura della CPU, il ciclo macchina e le funzionalità di esecuzione delle ISR (e di set/reset dei relativi flag di interrupt!) qualora dovesse affrontare la programmazione di un microcontrollore specifico di un certo *vendor* (con differenze, a volte, anche tra famiglie diverse dello stesso produttore).

B.2.2.6 Programmazione microcontrollori: strumenti e principi generali

In questa sezione verranno elencate delle linee guida di validità generale, cioè applicabili a qualunque dispositivo di qualunque produttore, per la programmazione di un microcontrollore.
Anzitutto, è necessario evidenziare le seguenti considerazioni:

- il microcontrollore costituisce il nucleo principale (spesso l'unico elemento) di un sistema *embedded*, raramente dotati di dispositivi di memoria come hard-disk o simili, comuni invece nei PC, e sui quali raramente vengono usati sistemi operativi per il supporto dei programmi utente o, comunque, eventuali sistemi operativi installati non sono standardizzati come quelli per PC (i.e. Microsoft Windows, Ubuntu/Debian Linux, ecc.);
- il programma del microcontrollore deve prevedere tutte le funzionalità di gestione diretta dell'hardware che in un PC, o comunque in un calcolatore dotato di sistema operativo, sono risolte dal sistema operativo stesso;
- il microcontrollore ha solitamente un proprio linguaggio Assembly molto specifico, per cui la stesura del codice va fatta rispettando la sintassi indicata nei relativi manuali di programmazione. Nel caso sia disponibile un compilatore per linguaggi di alto livello (es. BASIC o C), questo compilatore è normalmente proprietario e, pertanto, non gratuito, in quanto deve essere in grado di tradurre in modo ottimizzato il codice di alto livello nell'Assembly specifico[13];
- trovato l'ambiente di sviluppo software opportuno per il microcontrollore selezionato, occorre dotarsi di un dispositivo specifico, il vero e proprio **programmatore** hardware, per il caricamento del codice eseguibile sulla memoria "on-chip" del microcontrollore stesso, la quale può essere una ROM non riscrivibile, una EEPROM, una Flash memory, ecc.;

In sostanza, la programmazione di un microcontrollore richiede una scelta degli strumenti molto specifica, in funzione del particolare microcontrollore che si desidera utilizzare. Nel seguito, si

[13]Per le piattaforme PC esistono invece numerosi compilatori standardizzati, che permettono di generare un programma eseguibile in funzione del sistema operativo.

cercherà di descrivere i concetti basilari per la programmazione di un generico microcontrollore, utilizzando brani di codice in linguaggio C, essendo questo il linguaggio maggiormente utilizzato dai progettisti di sistemi embedded.

La struttura generale di un programma per un microcontrollore può essere schematizzata come segue:

1. **Inizializzazione**: in questa prima sezione del software, vengono impostate le **configurazioni degli SFR**, in base al funzionamento specifico richiesto per le periferiche integrate. Ad esempio, nel caso si debba eseguire operazioni periodiche con temporizzazioni predefinite, sarà necessario predisporre una periferica di conteggio (es. il Timer0 nell'8051) in modo che funzioni in modalità "timer" e che sia possibile gestire la condizione di roll-over in modo che essa avvenga con le tempistiche necessarie. In linguaggio C, le funzioni di inizializzazione devono essere eseguite immediatamente all'inizio della funzione `main` del programma:

```
void main(void)
{
  InitTimer0(); // Inizializzazione del Timer0
...
} //Fine del main
```

2. **Ciclo principale**: in questa sezione, viene programmato il compito vero e proprio del microcontrollore, vale a dire il controllo del sistema fisico ad esso collegato (tramite sensori/attuatori). Tipicamente, le istruzioni che costituiscono il programma principale devono essere eseguite ciclicamente per una ripetizione virtualmente infinita, in modo da mantenere sempre attivo il controllo in **tempo reale** del sistema. Ad esempio, con un linguaggio di alto livello come il C, si potrebbe programmare un loop infinito all'interno del `main()` (ovviamente dopo l'inizializzazione, del tipo:

```
void main(void)
{
  Initxxx();
..
  while(1)
  {
  // Istruzioni del programma principale
  ...
  } //Fine del while(1)
} //Fine del main
```

3. **Terminazione**: in questa sezione, tipicamente eseguita solo in caso di guasto o in fase di spegnimento del sistema di controllo, si eseguono eventuali operazioni necessarie alla "messa in sicurezza" del sistema controllato (es. spegnimento degli azionamenti elettrici ed attesa dei transitori di scarica di condensatori, ecc.). Nell'esempio precedente in linguaggio C, il codice relativo alla terminazione del programma, potrebbe essere inserito come segue:

```
void main(void)
{
..
  while(1)
  {
  // Istruzioni del programma principale
  ...
  if (SWITCH_OFF || CRITICAL_FAULT) break;
  } //Fine del while(1)
// Istruzioni di terminazione
...
} //Fine del main
```

4. **Interruzioni**: come detto, la condizione di overflow di una delle periferiche di conteggio (es. il Timer0 nell'8051), la ricezione di un dato dalla porta seriale ed eventuali impulsi esterni (es. sui pin P3.2 e P3.3 nell'8051) sono degli eventi che vengono tipicamente gestiti interrompendo il programma principale ed eseguendo specifiche routine di "risposta" all'interrupt (ISR). Nei

microcontrollori, l'interruzione del programma principale e la "chiamata" della routine di risposta sono effettuate in modo automatico. Ciascuna possibile causa di interrupt è associata ad una specifica locazione di memoria ROM, che conterrà le istruzioni eseguite dal processore nel ciclo macchina immediatamente successivo a quello nel quale si verifica l'evento di interruzione. Se il programma è scritto direttamente in Assembly, è abbastanza agevole organizzare le istruzioni in modo che quelle relative alla gestione di una interruzione siano associate agli indirizzi corretti, in quanto questo tipo di linguaggio fornisce proprio direttive specifiche per l'allocazione di memoria sia di dati che di istruzioni. Se invece si fa uso di un compilatore C o BASIC, è necessario che tale compilatore metta a disposizione delle **parole chiave specifiche** per l'identificazione di funzioni o procedure che non sono eseguite per chiamata, ma in relazione agli interrupt. Ad esempio, in un generico compilatore C per 8051, si potrebbe scrivere:

```
1 void timer0_isr(void) interrupt 0x0B
2 {
3 // Routine di gestione overflow Timer0
4 ..
5 }
```

supponendo che `interrupt 0x0B` sia la parola chiave ammessa per identificare in modo esplicito la funzione `timer0_isr()` come routine di risposta all'interrupt di overflow del Timer0 (locazione di memoria di indirizzo 0BH). Si noti che, sebbene lo spazio di memoria dedicato alle ISR sia in genere di dimensione limitata (es. 8 byte nell'8051), la funzione programmata in C non è soggetta a limitazioni di lunghezza, in quanto la corretta traduzione nelle istruzioni Assembly per salvataggio/ripristino dei registri (*context switch*) ed eventuale salto ad altre locazioni di memoria fuori dalla IVT (o IVA nei PIC) è risolta in fase di compilazione. Si noti inoltre che, in generale, le ISR non ammettono nè il passaggio di parametri nè un valore di ritorno, pertanto **l'unico modo** per avere lo scambio di informazioni tra il programma principale e le ISR è quello di definire delle **variabili globali** accessibili in entrambi i contesti.

Accorgimenti importanti nella programmazione ad interrupt

La programmazione delle ISR è uno degli aspetti fondamentali, insieme alla configurazione ottimale delle periferiche integrate tramite i relativi SFR, dello sviluppo di software per microcontrollori. Una corretta ed efficiente suddivisione tra il programma principale e le ISR delle operazioni richieste da una certa applicazione è infatti cruciale per sfruttare pienamente le risorse computazionali del dispositivo, spesso limitate per limitare anche i costi dell'hardware di progetto (i.e. risparmiare 1 € sul costo di un componente, in una scheda costruita in migliaia di unità, significa un ovvio risparmio di notevole entità...), evitando al contempo possibili errori e malfunzionamenti dovuti alla mancata risposta del sistema *embedded* agli stimoli ed eventi esterni (es. mancata ricezione di un byte su una linea di comunicazione ad alta velocità, perchè il relativo evento non è stato gestito prontamente).

Le linee guida per la buona programmazione (i.e. *best practices* o *coding styles*) di software per ambiente Personal Computer o simili spesso non sono altrettanto valide per la programmazione di sistemi *embedded*. Sebbene per quest'ultima esistano diversi fonti specifiche di letteratura tecnica, il contesto è ulteriormente complicato dal fatto che microcontrollori con architetture differenti e progettati da *vendor* diversi possono richiedere accorgimenti di programmazione ancora più specifici (i.e. si veda la differenza nella gestione degli interrupt tra Intel 8051 e Microchip PIC). Tuttavia, nel seguito verranno elencate alcune considerazioni sufficientemente generiche da poter essere utili in un ampio numero di applicazioni[14].

La prima regola della programmazione di una ISR si può riassumere con il motto: **keep it short and simple**. Le funzioni ISR, cioè, dovrebbero essere corte ed eseguire operazioni semplici, possibilmente limitate ad operazioni su singole variabili booleane o intere, in quanto il loro ruolo è appunto quello di gestione rapidamente eventi specifici e potenzialmente non predicibili. Ad esempio, la ISR può forzare una variabile *flag* al valore vero, se si è verificata una opportuna condizione

[14] I suggerimenti qui riportati sono tratti dal manuale del compilatore open-source e multi-target SDCC (http://sdcc.sourceforge.net/doc/sdccman.pdf, paragrafo 3.8), e dal seguente articolo in tre parti (link alle parti 2 e 3 in fondo alla pagina linkata): https://www.embedded.com/interrupts-short-and-simple-part-1-good-programming-practices/

associata all'evento di interrupt, la cui valutazione non può essere ritardata. Oppure, la ISR associata alla scadenza di un Timer può incrementare una o più variabili intere ausiliarie, per permettere al programma principale di gestire più operazioni periodiche con frequenze differenti, ma utilizzando un unico temporizzatore hardware (generalmente disponibili in numero limitato!). Infine, la ISR associata alla ricezione di un byte su una linea di comunicazione può trasferire tale byte in un buffer software di dimensione opportuna, in modo da garantire la gestione da parte del programma principale di ogni dato ricevuto (spesso le periferiche di comunicazione tipo UART o simili hanno buffer hardware di pochi byte, sovrascritti in caso di ricezioni multiple non gestite dal software).

Un altro aspetto cruciale nella programmazione delle ISR è il corretto uso delle **variabili globali**. Le ISR, infatti, non possono avere parametri nè restituire risultati se non attraverso aree di memoria condivise con il programma principale e, quindi, con ogni altra funzione del software, sia ISR che normali routine. Anzitutto, se si usa un compilatore C o C++, le variabili globali che possono essere modificate da una ISR devono essere dichiarate con parola chiave opzionale `volatile`, cioè ad esempio:

```
volatile int shared_var;
```

Tale opzione informa il compilatore che la variabile può cambiare stato anche al di fuori del contesto del programma principale o delle funzioni da esso richiamate, impedendo che il compilatore stesso possa erroneamente tradurre in Assembly alcune parti di codice cercando di ottimizzarne le prestazioni. Ad esempio, il seguente uso della variabile dichiarata come sopra:

```
int main()
{
   shared_var = 0;
   ...
   while(shared_var < 100)
       ;
   ..
}
```

potrebbe essere collegato al fatto che tale variabile viene incrementata da una ISR e che il ciclo nel `main()` serva per attendere la centesima esecuzione della ISR stessa. Tuttavia, il compilatore C potrebbe supporre (**erroneamente!**) che non essendoci altre assegnazioni a `shared_var` nel programma principale, la condizione del ciclo `while(..)` si possa tradurre in una condizione sempre vera, eliminando la valutazione dell'operatore di confronto per risparmiare istruzioni. La dichiarazione `volatile` evita l'applicazione di questa regola di ottimizzazione, che nel caso considerato determinerebbe un comportamento del software (insospettabilmente, per il programmatore) errato. Inoltre, una variabile globale modificabile da una ISR, NON dovrebbe essere utilizzata in più punti del programma principale, per evitare possibili comportamenti non deterministici causati da una scrittura da parte dell'ISR che avvenga tra due diversi punti di lettura nel `main()`. Ad esempio, nei due listati seguenti l'implementazione di sinistra può causare l'invio di un dato sbagliato, se viene assegnato dalla ISR il valore 1 alla variabile `Command`, **proprio mentre** il `main()` sta valutando la condizione `else if(Command == 2)`: in tal caso, infatti, si eseguirebbe l'ultimo blocco di `else`, inviando Data 3 anzichè Data 1 come richiesto. Il comportamento del codice di destra è invece esente da questo difetto, grazie alla copia locale nel `main()` della variabile condivisa.

Caso A: sbagliato!

```
volatile char Command;

int main()
{
    // Setup UART and enable interrupt
    ...
    while(1){
        if(Command == 1){
            // Send Data 1
        }
        else if(Command == 2){
                // Send Data 2
        }
        else{
            // Send Data 3
        }
    }
}

void UART_ISR(void) interrupt xxx
{
    Command = UART_RX_BUF;
}
```

Caso B: corretto

```
volatile char Command;

int main()
{
    char LocalCmd;

    // Setup UART and enable interrupt
    ...
    while(1){
        LocalCmd = Command;
        if(LocalCmd == 1){
            // Send Data 1
        }
        else if(LocalCmd == 2){
                // Send Data 2
        }
        else{
            // Send Data 3
        }
    }
}

void UART_ISR(void) interrupt xxx
{
    Command = UART_RX_BUF;
}
```

Infine, è bene considerare sempre che l'accesso ad una variabile con dimensione in byte maggiore di quella elaborata da **operazioni atomiche** della CPU nel microcontrollore programmato (es. variabile intera a 16 bit con una CPU a 8 bit o variabile a 32 bit con una CPU a 16 bit), è invece una **operazione non atomica** e, quindi, potenzialmente interrompibile da una ISR durante l'esecuzione delle istruzioni Assembly associate alla manipolazione parziale della variabile stessa. In altre parole, anche la semplice copia tra due variabili `int` in linguaggio C può essere corrotta, se la CPU è ad 8 bit ed è interrotta dall'ISR nel momento sbagliato. Per evitarlo, è buona norma sfruttare le opzioni specifiche per **disabilitare gli interrupt** per la CPU utilizzata (solitamente collegate al set/reset di opportuni bit in un SFR) in corrispondenza dell'esecuzione di operazioni di lettura di variabile condivise con ISR. Nell'ipotesi che allo scopo siano definite le funzioni `DisableInterrupts()` ed `EnableInterrupts()`, i due listati seguenti mostrano il modo sbagliato per accedere dal `main()` al valore a 16 acquisito tramite ADC ad interrupt, su un microcontrollore a 8 bit (a sinistra), ed il corrispondete modo corretto (a destra):

Caso A: sbagliato!

```
volatile int ADCValue;

int main()
{
   int LocalADCVal;
   // Setup ADC and enable interrupt
   ...
   while(1){
        LocalADCVal = ADCValue;
        ...
   }
}

void ADC_ISR(void) interrupt xxx
{
   // 16 bit A/D conversion is stored in
   //            ADC_BUF1 (upper byte)
   //        and ADC_BUF0 (lower byte)
   ADCValue = (ADC_BUF1<<8)+ADC_BUF0;
}

```

Caso B: corretto

```
volatile int ADCValue;

int main()
{
   int LocalADCVal;
   // Setup ADC and enable interrupt
   ...
   while(1){
        DisableInterrupts();
        LocalADCVal = ADCValue;
        EnableInterrupts();
        ...
   }
}

void ADC_ISR(void) interrupt xxx
{
   // 16 bit A/D conversion is stored in
   //            ADC_BUF1 (upper byte)
   //        and ADC_BUF0 (lower byte)
   ADCValue = (ADC_BUF1<<8)+ADC_BUF0;
}

```

Per concludere, gli accorgimenti citati sono solo alcune tra le soluzioni possibili ai problemi tipici della programmazione di ISR per microcontrollori. Ad esempio, non si sono trattati gli aspetti legati all'eventualità che le ISR possano essere interrompersi a vicenda (*nested interrupts*, v. Figura B.2.20), anche perchè come si è visto tale possibilità non è supportata da tutti i microcontrollori. Per uno specifico dispositivo programmato con uno specifico compilatore, il riferimento migliore per avere le soluzioni ottimali è certamente il manuale del compilatore stesso, che includerà sempre almeno un capitolo intero dedicato all'argomento "programmazione ISR".

Operazioni tipiche di base: temporizzazione e *debouncing* degli input

I sistemi di elaborazione per il controllo devono tipicamente eseguire le proprie operazioni in modo opportunamente temporizzato. Misurare il passaggio di un determinato periodo di tempo, in modo preciso, è quindi una delle funzionalità fondamentali nella programmazione di un microcontrollore. Tipicamente, tale funzionalità si può realizzare in due modi:

1. tramite funzioni di *ritardo* o *attesa*, che bloccano l'esecuzione della parte ciclica del programma principale, per un tempo prefissato;

2. tramite la configurazione di Timer hardware e delle relative ISR.

Come esempio del primo caso, è riportato nel seguito un listato nel quale è definita una funzione bloccante, appunto, il cui contenuto consiste nell'esecuzione per un numero di ripetizioni opportuno di una istruzione *null*, che occupa cioè la CPU per un ciclo macchina, ma senza che venga eseguita alcuna elaborazione di dato. Tale funzione è poi utilizzata nel `main()` per fare accendere e spegnere un led con un periodo totale del lampeggio di 1 secondo. Lo svantaggio principale di tale soluzione è ovviamente che la CPU è di fatto inutilizzata per la maggior parte del tempo, senza considerare che una taratura precisa del numero di cicli necessari, all'interno della funzione `DelayMs()`, per ottenere le temporizzazioni desiderate è normalmente empirica.

```
// 120 machine cycles = 1 ms (to be adjusted for specific microcontroller!)
void DelayMs(unsigned int millisecs){
   unsigned int x,y;

   for(x = 0; x < millisecs; x++){
       for(y = 0; y < 120; y++)
           Nop(); // null operation, forcing one machine cycle
   }
}

void main(){

  InitIOPins();

  while(1){
     LED = 1;
     DelayMs(500);
     LED = 0;
     DelayMs(500);
  }
}
```

Con la seconda soluzione, invece, si possono sfruttare pienamente le capacità della CPU e, soprattutto, le periferiche progettate in modo specifico per tali scopi: i Timer. Il listato seguente mostra come ottenere la commutazione degli 8 bit della porta B di un PIC con una frequenza opportuna, determinata dall'inizializzazione del Timer0 al valore di 98 e con un **prescaler**, cioè un divisore della frequenza del ciclo macchina della CPU per rallentare l'incremento del timer, di 1:32 (i.e. timer 32 volte più lento della CPU). Grazie al prescaler, è possibile misurare più precisamente anche periodi di tempo relativamente lunghi (rispetto al ciclo macchina, di pochi nanosecondi). L'impostazione del valore iniziale del timer, ripetuta ad ogni esecuzione della ISR, permette invece di calibrare il periodo di esecuzione della ISR stessa, corrispondente all'evento di overflow di TMR0L (quindi, ogni 256-98=156 cicli).

```
void interrupt ISR(){

  if (TMR0IF && TMR0IE) {  //Check if Timer0 has caused the interrupt

      PORTB = ~PORTB; //Toggle PORT to generate square wave

      TMR0IF = 0;      // clear interrupt flag

      TMR0L = 98;      // Preload timer0
  }
}

void main(){

  TMR0L = 98;    // Timer0 preload value
  T0CON = 0xC4; // = 0b11000100  Set TMR0 to 8bit mode and prescaler to 1:32

  GIE = 1;       // Enable global interrupt
  TMR0IE = 1;    // Enable Timer0 interrupt
    ...
    while(1){
      ..
    }
}
```

Un ulteriore esempio di operazione tipicamente legata alla gestione del tempo, ma prettamente con l'obiettivo di attendere un certo periodo e, quindi, riconducibile all'uso della `DelayMs()` precedentemente descritta, è il cosiddetto *debouncing* degli input collegati a contatti elettromeccanici (es. pulsanti, sensori di finecorsa, ecc.). Tali dispositivi, i cui tipici schemi di interconnessione ad un input pin sono mostrati in Figura B.2.22, sono spesso affetti dal problema della commutazione non

stabile a seguito della loro attivazione o disattivazione. Come mostrato in Figura B.2.23, infatti, lo stato logico del pin connesso ad uno di tali interruttori potrebbe *rimbalzare* (in inglese, *to bounce*) tra vero e falso per un breve periodo di tempo, prima di poter essere considerato valido.

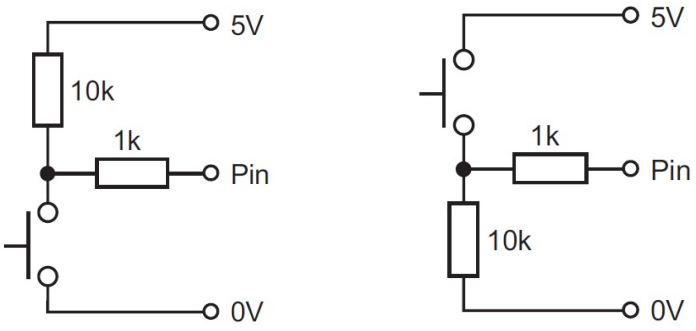


Figura B.2.22: Connessione a un *input pin* di interruttori elettromeccanici: con *pull-up* (a sinistra) o *pull-down* (a destra). Si veda http://www.picaxe.com/docs/picaxe_manual3.pdf per una panoramica di circuiti di interfaccia tipici per microcontrollori.

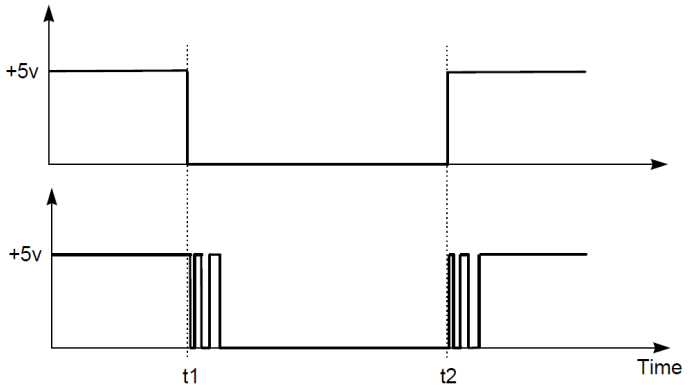


Figura B.2.23: Segnale logico ideale (in alto) ed affetto da *bouncing* (in basso), tipico di interruttori elettromeccanici

Per garantire la lettura dello stato logico quando questo è effettivamente valido, è bene eseguirla due volte a distanza di qualche centinaio di millisecondi. Il listato riportato nel seguito mostra la logica di debouncing per un pulsante con circuito di interfaccia dotato di resistenza di *pull-up* (v. Figura B.2.22, a sinistra), per intercettare correttamente l'evento di pressione del pulsante (i.e. stato logico falso).

```
// Input_Switch has a pull-up resistor, so is false when pressed
if(!Input_Switch){

   DelayMs(100); // switch debouncing delay

   if(!Input_Switch){
      // process switch pressed event
      ....
   }
}
```

Esecuzione di *task* periodici: Timers/ISR vs. RTOS

La sotto-sezione precedente ha evidenziato come la programmazione di operazioni periodiche sia di fatto l'essenza dello sviluppo di software per microcontrollori. Al di là degli esempi didattici qui

mostrati, una applicazione industriale anche di complessità limitata può richiedere l'esecuzione di diversi compiti (o *task*) a periodicità differente:

- più controllori in cascata (es. posizione/velocità/corrente di un motore elettrico), da eseguire con frequenza crescente dal loop esterno verso l'interno;
- *debouncing* di molteplici input;
- segnalazioni con molteplici LED lampeggianti a frequenze differenti;
- gestione di linee di comunicazione differenti a varie velocità di trasmissione.

In un contesto simile, associare ciascuna operazione periodica alla temporizzazione di uno specifico Timer può essere impossibile (es. microcontrollore con 5 Timer e 12 task periodici da eseguire). Una tipica soluzione al problema è quella di sviluppare (o utilizzare, se disponibile come libreria software compatibile) uno *scheduler* di processi, anche con una logica semplice e di complessità minima, in grado di eseguire un numero indefinito di task periodici configurando però un unico Timer "base". Con tale approccio, la scadenza del timer base definisce il "tick" elementare dello scheduler, il cui compito nell'ipotesi più semplice è quello di contare i "tick" e di verificare in base a tale conteggio quali task siano da eseguire (eventualmente anche secondo una priorità programmabile).

Nel seguito sono riportate alcune parti di codice di un possibile scheduler di processi periodici[15]. Anzitutto, ad ogni task è associata una struttura dati del tipo:

```
typedef struct {
  uint32_t delay;
  uint32_t period;
  uint8_t run;
  void (*exec)(void *handle);
  void *handle;
} tTask;
```

che contiene le impostazioni di periodo e/o ritardo di esecuzione, il flag settato dallo scheduler quando il task è pronto per essere eseguito, il puntatore `exec` alla funzione da eseguire quando il flag `run` del task è settato e il puntatore `handle` ad una eventuale lista di parametri da passare a tale funzione. Sulla base di questa struttura, lo scheduler si compone delle seguenti funzioni:

```
/* Inizializzazione dello scheduler e dell'array di task */
void schInit(){ ... }

/* Creazione di un task */
void schAdd(void (*exec)(void *), void *handle, uint32_t delay, uint32_t period)
{ ... }

/* Eliminazione di un task */
void schDel(void (*exec)(void *), void *handle){ ... }

/* Aggiornamento della situazione dei task e settaggio dei relativi flag ''run'', da
    eseguire nella ISR associata al timer base (i.e. tick dello scheduler) */
void schUpdate(){
  for (uint8_t i = 0; i < MAX_NUM_OF_TASKS; i++) {
    if (tasks[i].exec != NULL) {
      if (tasks[i].delay == 0) {
        tasks[i].delay = tasks[i].period;
        tasks[i].run++;
      } else {
        tasks[i].delay--;
      }
    }
  }
}

```

[15] La logica dello scheduler presentato è tratta dal libro "Patterns for Time-Triggered Embedded Systems" di Michael J. Pont, scaricabile gratuitamente dal link https://www.safetty.net/publications/pttes. Una introduzione più breve e generale dello scheduler proposto da Pont è fornita a questo link https://maker.pro/custom/tutorial/a-simple-time-trigger-scheduler-for-stm32-and-msp430-projects.

```
25 /* Corpo principale dello scheduler, da chiamare ciclicamente (es. in un loop
       infinito all'interno del main()) */
26 void schExec(){
27   for (uint8_t i = 0; i < MAX_NUM_OF_TASKS; i++) {
28     if (tasks[i].exec != NULL && tasks[i].run > 0) {
29       tasks[i].run--;
30       tasks[i].exec(tasks[i].handle);
31       if (tasks[i].period == 0) {
32         tasks[i].exec = NULL;
33       }
34     }
35   }
36 }
```

Se oltre a queste funzioni di temporizzazione relativamente semplici, sono necessarie sincronizzazioni tra i task (es. locazioni di memoria condivisa ad accesso controllato, segnalazioni tra processi con *semafori*, ecc.) o altre funzionalità più complesse (es. gestione dell'intero stack TCP/IP o di una memoria con file system), si può fare ricorso a veri e propri sistemi operativi, detti **Real-Time Operating Systems (RTOS)**. Per il contesto dei microcontrollori sono disponibili moltissimi RTOS, sia open-source che commerciali, quali ad esempio: **FreeRTOS**[16], **eCOS**[17], **ChibiOS**[18], per citarne solo alcuni tra quelli open-source/freeware e multi-target.

In-Circuit Programmer/Debugger vs. Bootloader

Una volta sviluppato con unn opportuno linguaggio (i.e. C o Assembly) e compilato il programma per un microcontrollore, quello che si ottiene è un file contenente il codice macchina, tipicamente memorizzato in un formato **binario**, oppure in formato testuale con estensione .HEX contenente il valori **esadecimali** dei byte di Code Memory. Il contenuto di tali file deve trasferito dal Personal Computer utilizzato per lo sviluppo del software alla Code Memory stessa del microcontrollore. Tale operazione richiede un dispositivo specifico per la scrittura di memorie non volatili, in funzione della tecnologia contenuta nel chip (EEPROM o Flash). Fino ad alcuni anni fa, inoltre, la scrittura della Code Memory era possibile solo fissando il microcontrollore su una scheda elettronica dedicata, differente da quella di destinazione finale o *target*. Oggi, la modalità di programmazione più diffusa per questo tipo di memorie è quella cosiddetta **In-Circuit**, cioè che può essere fatta sul microcontrollore già installato sulla scheda target ed è gestita direttamente dalla sua CPU tramite operazioni specifiche. L'esecuzione di tali operazioni deve però essere forzata (es. tramite l'applicazione di una tension di alimentazione più alta di quella nominale) dal dispositivo di programmazione esterno già citato. In tale contesto, tale dispositivo è definito **In-Circuit (Serial) Programmer (ICP o ICSP)**.

Il ruolo dell'ICP/ICSP è molto importante anche per le fasi di sviluppo e test del software stesso. Infatti, nella modalità di programmazione In-Circuit il microcontrollore può anche essere forzato ad eseguire il codice nella Program Memory in modo controllato dall'ICP, che può:

- riavviarlo (i.e. forzare il *reset*);
- bloccarlo (i.e. forzare un *breakpoint*);
- farne eseguire una singola istruzione alla volta (i.e. esecuzione *step-by-step*).

Ovviamente, tali operazioni sono gestite tramite il Personal Computer sul quale è installato lo strumento di sviluppo integrato (**Integrated Development Environment, IDE**) per quel microcontrollore e l'ICP diventa di fatto uno strumento di **debug** (**In-Circuit Debugger, ICD** o **In-Circuit Emulator, ICE**). Nella Figura B.2.24 è mostrato a titolo di esempio il principio di funzionamento del dispositivo Real ICE di Microchip, collegabile ad un PC tramite connessione USB e ad una scheda target con microcontrollori di qualsiasi famiglia Microchip tramite connessione proprietaria a 6 pin.

Un'alternativa all'uso del dispositivo di programmazione/debug In-Circuit, il cui costo è solitamente di alcune centinaia di €, è costituito dall'installazione sulla scheda target di un firmware

[16] https://www.freertos.org/about-RTOS.html
[17] https://ecos.sourceware.org/about.html
[18] https://www.chibios.org/dokuwiki/doku.php

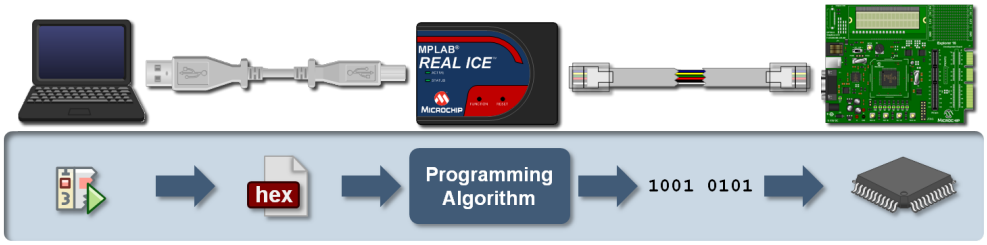


Figura B.2.24: In-Circuit Programmer/Debugger: principio di funzionamento

predisposto al trasferimento nella Code Memory di dati ricevuti tramite una normale interfaccia di comunicazione seriale, sempre sfruttando il fatto che la CPU del microcontrollore può eseguire, se posta in un certa modalità operativa, anche la scrittura sulla propria memoria EEPROM o Flash. Il firmware in questione viene generalmente definito **(boot)loader**, in quanto predisposto per essere eseguito al reset del microcontrollore e per rimanere un certo periodo di tempo in attesa dei dati da caricare in Code Memory, terminato il quale viene avviata l'esecuzione del codice *applicativo*. L'installazione di un bootloader sulla scheda target ne permette la programmazione anche da parte dell'utilizzatore finale, che può quindi evitare l'oneroso acquisto di un ICP/ICSP. Tale strategia è quindi particolarmente diffusa per le schede a microcontrollore pensate per un uso hobbystico, come ad esempio quelle della famiglia **Arduino**, la cui modalità di programmazione hardware è mostrata in Figura B.2.25.

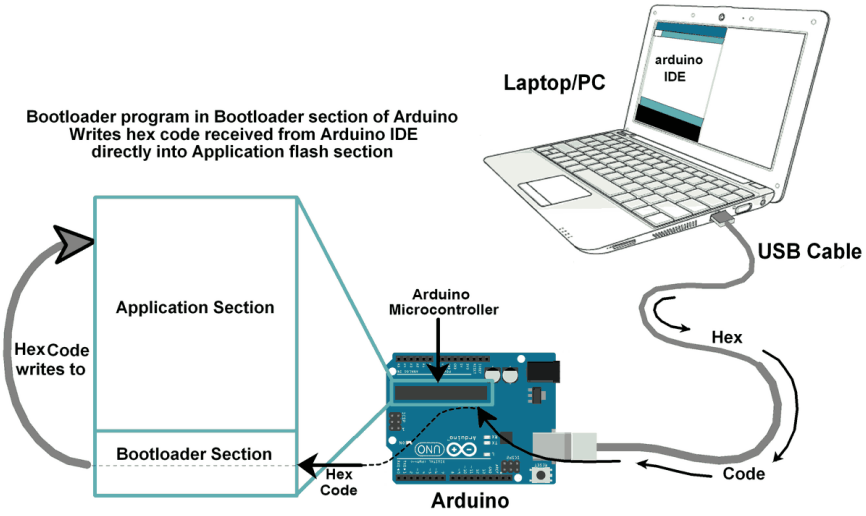


Figura B.2.25: Programmazione di microcontrollori tramite *bootloader*

B.2.3 Il microcontrollore *per tutti*: Arduino

Arduino è il nome di un progetto *open-source*, avviato nel 2005 da un team dell'Interaction Design Institute di Ivrea (fondato da Olivetti e Telecom Italia), il cui obiettivo era sviluppare piattaforme a basso costo e facili da usare per la protipazione rapida di applicazioni di elettronica e controllo. Vista l'attuale diffusione e disponibilità commerciale delle schede elettroniche a microcontrollore basate sulla piattaforma Arduino, sia nelle sue varianti "ufficiali" che nelle versioni classificabili come "concorrenti" (quando non addirittura "cloni"), l'obiettivo iniziale del progetto Arduino si può dire ampiamente raggiunto! Basta fare una rapida ricerca sul web o sfogliare riviste specializzate

(ma anche solo divulgative) di elettronica e informatica per capire che le schede Arduino sono usate da migliaia di hobbisti, designer e creativi in genere per realizzare ogni tipi di oggetto o applicazione che richieda interattività tra un sistema programmabile e l'ambiente che lo circonda.

Dal punto di vista tecnologico, le schede Arduino non contengono altro che un microcontrollore e pochi altri dispositivi ausiliari, tra i quali il principale è quello per la gestione della comunicazione USB. Tramite questa connessione, infatti, viene trasferito direttamente nella memoria del microcontrollore il programma utente, senza la necessità di altri programmatori hardware. Ciò è possibile grazie ad un *bootloader* pre-caricato sul microcontrollore dell'Arduino. La più diffusa tra le schede Arduino è ad oggi la versione **Uno**[19], mostrata in Figura B.2.26.

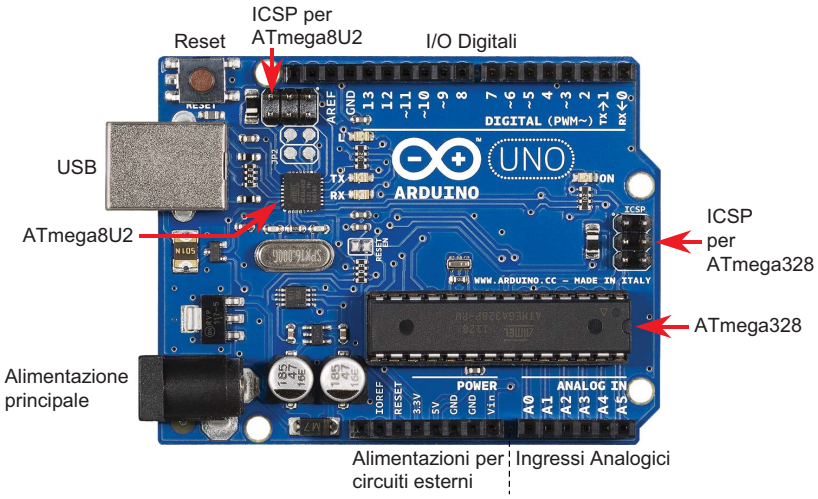


Figura B.2.26: Scheda elettronica Arduino Uno

Come si può notare, Arduino Uno contiene in effetti due microcontrollori: l'unità principale (*target* della programmazione) è un chip Atmel ATmega328 della famiglia AVR a 8 bit, mentre il chip Atmel ATmega8U2, ancora un AVR a 8 bit, è pre-programmato per la gestione della comunicazione USB come *device* Serial Port Profile (SPP). Con questo profilo, Arduino è riconosciuto da un Personal Computer (*USB Host*) come una porta seriale standard. Entrambi i chip ATmega sono quindi pre-programmati dal produttore della scheda (quello principale con un *bootloader*), ma potrebbero anche essere ri-programmati dall'utente finale attraverso i connettori **ICSP (In-Circuit Serial Programming)**. Questa ultima operazione però richiede un dispositivo ausiliario per la programmazione hardware (es. AVRISP prodotto da Atmel, oppure una seconda scheda Arduino[20]).

Le schede Arduino sono inoltre facilmente espandibili tramite i connettori a pettine laterali. La distanza tra questi connettori non è compatibile con il passo di una normale scheda *millefori* per prototipazione elettronica, ma è possibile reperire sul mercato di componenti per hobbistica un grande numero di schede di espansione specifiche (es. con circuiti di potenza per motori elettrici o relè, con schede di memoria SD, con moduli per la comunicazione USB Host o Ethernet/WiFi o GSM, ecc.), denominati *Shield*. Poichè anche il progetto hardware di Arduino è open-source, esistono molte varianti (non "ufficiali") basate su microcontrollori diversi dagli AVR (es. schede Nucleo di ST Microelectronics, con STM32) compatibili con gli shield nel formato di Arduino.

Le caratteristiche hardware degli AVR sono molto simili a quelle di altri microcontrollori ad 8 bit "moderni" e possono essere riassunte come segue:

- CPU RISC con architettura Harvard (bus separato per memoria dati e memoria codice).
- Porte di I/O con configurazione tri-state.

[19] Maggiori dettagli su tutte le schede della famiglia Arduino e sul relativo software di programmazione sono disponibili alla pagina http://arduino.cc/

[20] Si veda il tutorial alla pagina http://arduino.cc/en/Tutorial/ArduinoISP

- Convertitore A/D a 10 bit a 6 ingressi multiplexati.
- Due timer a 8 bit e un timer a 16 bit, con funzionalità di *Input Capture* e generazione PWM.
- Moduli per comunicazione seriale sincrona (SPI) e asincrona (UART).
- Interrupt Controller con 26 possibili eventi associabili ad altrettante ISR differenti.

Il vero motivo del successo di Arduino non è però legato alle caratteristiche hardware, che sono infatti analoghe a quelle delle schede dimostrative o di prototipazione rapida da sempre distribuite dai principali produttori di microcontrollori o DSP/DSC, e neppure (o meglio non solo) al costo competitivo (ad inizio 2022, il prezzo "ufficiale" di Arduino Uno Rev3 è 20 Euro + IVA). La grande diffusione di Arduino si spiega senz'altro con la facilità di uso del suo ambiente di programmazione, **Arduino IDE**, e con la vastità di librerie incluse che rendono totalmente *invisibile* allo sviluppatore i dettagli hardware del microcontrollore installato. Tramite Arduino IDE infatti, non è necessario conoscere le definizioni degli SFR del chip ATmega328 per configurare le porte, effettuare le acquisizioni di segnali analogici o inviare dati via seriale. La Tabella B.2.2 riassume alcune tra le funzioni più usate nella programmazione di Arduino, che evidenziano la mancanza di riferimenti ai dettagli delle periferiche integrate nel microcontrollore AVR, come ADC o modulatori PWM (es. l'uscita PWM è in effetti vista come un output "analogico").

Funzione	Esempio	Descrizione
`pinMode`	`pinMode(8,OUTPUT);`	Imposta il pin 8 (indicazione numerica del connettore laterale Arduino, NON del chip ATmega328) come OUTPUT, con possibili alternative: INPUT o INPUT_PULLUP (tutte le porte di I/O dell'ATmega328 integrano resistenze di pull-up abilitabili via SFR).
`digitalWrite`	`digitalWrite(8,HIGH);`	Imposta al valore logico alto il pin 8
`digitalRead`	`in_switch = digitalRead(9);`	Legge il valore logico del pin 9 e lo assegna alla variabile `in_switch`
`pulseIn`	`time_in = pulseIn(8,HIGH);`	Calcola la durata in microsecondi dal prossimo fronte di salita del pin 8 al successivo fronte di discesa
`analogRead`	`tmp = analogRead(0);`	Legge il valore del segnale analogico A0, compreso tra 0 e 1023 (ADC dell'AVR a 10 bit)
`analogWrite`	`analogWrite(9,127);`	Genera un segnale PWM sul pin 9 (deve essere un pin con associato il simbolo ∼), con Duty Cycle al 50% (il Duty Cycle 100% corrisponde a 255)

Tabella B.2.2: Alcuni comandi base per la programmazione di Arduino

Un programma per Arduino (definito *sketch*) è costituito da un file in linguaggio C++, sebbene debba avere estensione `.ino`, che deve contenere la definizione di almeno due funzioni:

1. `void setup()`: funzione eseguita una sola volta in seguito al reset del microcontrollore, deve contenere le configurazioni di base dell'hardware.
2. `void loop()`: funzione eseguita ciclicamente in modo perpetuo, contenente la logica di controllo dell'applicazione.

Oltre a queste due funzioni, è possibile definire delle funzioni di risposta ad interrupt causati da:

- fronti di salita/discesa di pin digitali; la funzione ISR può essere una qualunque funzione utente, associata al pin e al tipo di evento di interrupt con:

```
attachInterrupt(pin_n,nome_ISR,FALLING/RISING);
```

- overflow del Timer1; per associare una funzione ISR al timer è necessario includere esplicitamente nello sketch la libreria `TimerOne.h` e inizializzare il periodo del timer come segue:

```
Timer1.initialize(500); // in microsecondi
Timer1.attachInterrupt(nome_ISR);
```

  Si noti che l'uso del Timer1 impedisce l'uso delle funzioni `analogWrite()` sui pin 9 e 10, ma la libreria `TimerOne.h` include anche delle funzioni aggiuntive per configurare esplicitamente tali pin come generatori di segnali PWM.

B.2.3.1 Lo strato *nascosto* di Arduino IDE

L'ambiente di programmazione Arduino IDE ha una interfaccia utente molto semplice, che integra poco più di un editor di test ed i comandi per compilare il programma e trasferirlo sulla scheda target. Non esistono funzionalità di simulazione o di debug del codice (es. esecuzione bloccata tramite *breakpoint* o esecuzione istruzioni *step-by-step*). In realtà, Arduino IDE include molti più strumenti di quanto sia visibile dall'interfaccia utente (mantenuta volutamente scarna ed essenziale). Anzitutto, nella cartella di installazione del programma[21] è presente una sotto-cartella `../hardware/arduino/avr/cores/arduino`. Quest'ultima contiene i sorgenti di molte delle librerie disponibili dall'utente e anche molti file dei quali l'utente può tranquillamente ignorare l'esistenza. Tra questi però, salta all'occhio la presenza di un file `main.cpp`, il cui contenuto è particolarmente interessante:

```
#include <Arduino.h>
...
int main(void)
{
  init();

  #if defined(USBCON)
  USBDevice.attach();
  #endif

  setup();

  for (;;) {
       loop();
       if (serialEventRun) serialEventRun();
  }

  return 0;
}
```

Come si può notare, la struttura di questo file è analoga a quella presentata come riferimento generale nella Sezione B.2.2.6. La funzione `setup()` richiamata nel `main()` è esattamente quella programmata dall'utente tramite lo sketch, così come la funzione `loop()`, richiamata all'interno di un ciclo infinito `for(;;)` funzionalmente equivalente a `while(1)`. La prima funzione eseguita dal programma è però una funzione `init()`, non definita dall'utente ma "nascosta" tra i sorgenti predefiniti di Arduino IDE, così come le funzioni per la gestione della connessione USB[22] e la gestione ad evento della porta seriale. Il file `main.cpp` viene associato al contenuto dello sketch e poi trasformato in codice binario per AVR grazie al compilatore **AVR-GCC**, eseguito in modo nascosto da Arduino IDE. Il flusso completo di operazioni per la programmazione di Arduino è quindi schematizzabile come in Figura B.2.27.

[21] Si fa qui riferimento alla versione 1.8.11

[22] **NOTA BENE**: in questo caso si tratta della connettività USB gestita direttamente dal microcontrollore principale, che non è supportata dalla scheda Arduino Uno, ma ad esempio dalla variante Arduino Leonardo.

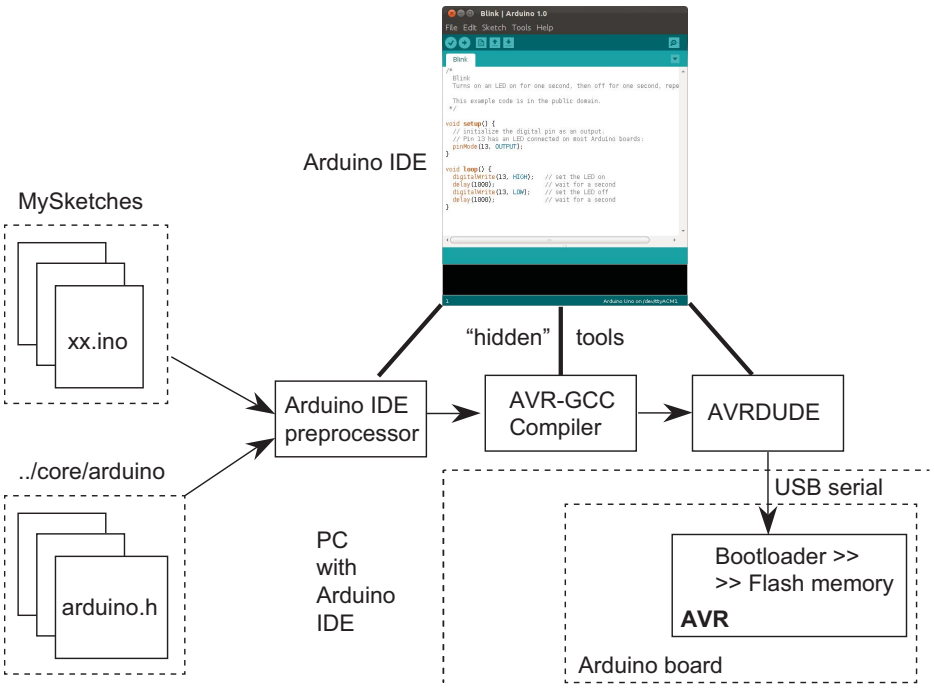


Figura B.2.27: Diagramma funzionale della *toolchain* di Arduino

Si noti che il livello di astrazione del processo di compilazione in Arduino IDE permette anche di rendere la procedura indipendente dallo specifico microcontrollore installato sulla scheda. Infatti, oltre alle varianti con AVR diversi da ATmega328, la più recente **Arduino Due** utilizza un microcontrollore ARM Cortex-M3 di Atmel, che richiede una struttura molto più complessa delle funzioni di gestione dell'hardware ed il compilatore **ARM-GCC**, entrambi peraltro invisibili dall'interfaccia utente.

B.2.3.2 I limiti dell'approccio Arduino

L'ambiente Arduino IDE e le relative librerie di alto livello sono strumenti generalmente sufficienti per realizzare moltissime applicazioni di controllo, purchè queste non richiedano prestazioni critiche o funzionalità supportate dalle periferiche hardware dei microcontrollori, ma non dalle librerie. Il principale limite delle librerie di Arduino è infatti il sacrificio delle prestazioni in favore della facilità d'uso e versatilità. Una operazione come la misurazione di durata per un impulso digitale, fondamentale ad esempio per il calcolo della velocità di un motore elettrico con encoder incrementale, viene implementata dalla funzione `pulseIn()` con una procedura **bloccante** (la funzione restituisce il valore solo **dopo** che l'impulso digitale è effettivamente terminato) e che NON fa uso delle funzionalità di *Input Capture* supportate direttamente via hardware dai timer sia degli AVR che degli ARM Cortex-M3. Anche l'operazione più banale per un microcontrollore, cioè settare il valore di una uscita digitale, è implementata in modo molto inefficiente dalle librerie di Arduino IDE. Si consideri ad esempio lo sketch:

```
byte state = 0;

void setup()
{
  pinMode(10,OUTPUT);
  while(true)
  {
    digitalWrite(10,state);
    state = !state;
  }
```

```
}

void loop(){}
```

che non fa altro che generare un segnale ad onda quadra direttamente nella funzione di `setup()` (la funzione `loop()` non contiene istruzioni). Misure effettuate con l'oscilloscopio e con una scheda Arduino Uno, dimostrano che la frequenza dell'onda quadra così ottenuta è di 86.39 kHz. Se invece di usare la funzione di libreria si accedesse direttamente agli SFR dell'ATmega328[23], il codice sarebbe:

```
void setup()
{
  DDRB = 0x04; // DDRB: Data Direction Register porta B
  // Pin 3 della porta B -> Digital I/O 10 di Arduino Uno
  while(true)
  {
     PORTB = 0x04; // Set pin 3
     PORTB = 0x00; // Reset all pins
  }
}

void loop(){}
```

In questo caso, la frequenza misurabile con oscilloscopio dell'onda quadra sarebbe 3.97 MHz, circa **46 volte maggiore** di quella ottenuta con l'uso della funzione di libreria `digitalWrite()`. D'altra parte, il rovescio della medaglia è che questa seconda versione del codice richiede di conoscere lo schema elettrico dell'Arduino Uno, dal quale è possibile ricostruire quale pin dell'ATmega328 sia collegato al pin 10 degli I/O digitali della scheda (la cui numerazione NON corrisponde a quella del chip ATMega328, ma è standardizzata dal *pinout Arduino* usato anche da schede con microcontrollore differente), senza contare che il codice basato sugli SFR dell'ATmega328 non è ovviamente portabile sulle schede di nuova generazione basate su ARM Cortex-Mx (al contrario di quello basato sulle librerie Arduino).

Si noti che l'accesso diretto agli SFR per la configurazione e la gestione delle periferiche integrate nel microcontrollore di Arduino permetterebbe anche la configurazione di un timer con modalità *Input Capture* e gestione ad interrupt (tramite la specifica ISR dell'evento di Capture, NON supportata però dalle funzioni `attachInterrupt()` viste in precedenza) della lettura del periodo misurato. Il codice con riferimenti specifici agli SFR è compilabile da Arduino IDE, ma ovviamente risulterebbe di difficile comprensione per un utente con poca esperienza nella programmazione di sistemi *embedded*.

Infine, si è citato in precedenza l'**assenza di strumenti di debug** del codice in Arduino IDE. Tali strumenti infatti richiedono:

1. Un dispositivo programmatore/debugger collegato al microcontrollore tramite il connettore ICSP.

2. Un ambiente di sviluppo completo di comandi per la gestione di breakpoint e modalità di esecuzione *step-by-step*, interfacciabile con il tool ICSP (es. **Microchip Studio**, precedentemente noto come Atmel Studio, strumento unico sia per AVR che ARM Cortex-Mx prodotti da Atmel/Microchip).

B.2.4 Digital Signal Processor (DSP) e Digital Signal Controller (DSC)

Il dispositivi classificati come DSP o DSC rappresentano una estrema specializzazione architetturale e funzionale dei sistemi di elaborazione, progettata per processare, in formato digitalizzato, segnali di qualsiasi natura (comunicazioni elettriche, sensori di misura, ecc.) ad un rapporto costo/prestazioni il più favorevole possibile. Questo tipo di componente è utilizzato in prevalenza nei campi in cui l'esecuzione estremamente veloce di elaborazioni complesse su segnali è un requisito importante, come ad esempio telefonia mobile, audio e video digitale e, soprattutto in tempi recenti, nei settori

[23] Si vedano per i dettagli il datasheet dell'ATmega328 alla pagina https://www.microchip.com/wwwproducts/en/atmega328 e lo schematico hardware di Arduino Uno

del controllo automatico in cui la banda passante dell'anello di controllo deve essere molto elevata (azionamenti di motori elettrici, robotica). Nelle recenti offerte commerciali dei principali fornitori di componenti elettronici, il termine DSP viene comunemente usato per indicare processori per applicazioni audio/video, che NON integrano molte periferiche ausiliare (spesso neanche i convertitori A/D e D/A) se non timer e memorie, mentre il termine DSC viene riferito ai processori per applicazioni di controllo, con una ricca dotazione di periferiche integrate funzionalmente analoghe a quelle precedentemente descritte per i microcontrollori (i.e. ADC, Input Capture, generatori PWM, contatori per Encoder Incrementali, ecc.). Nel seguito, si farà pertanto uso del termine DSP, qualora il contesto descriva aspetti generici architetturali, e DSC laddove si voglia evidenziare il maggiore interesse per questi processori nelle discipline controllistiche.

L'architettura interna di un generico DSP/DSC può essere schematizzata come in Figura B.2.28. Come si vede, le componenti integrate sul chip sono analoghe a quelle analizzate durante la descrizione dei microcontrollori. Inoltre, è importante osservare che per i DSP il **bus dati** e il **bus istruzioni** sono sempre **separati**. Come descritto in precedenza, questa tipologia di architettura, detta di Harvard, ha il notevole vantaggio di ridurre i tempi di esecuzione per istruzione rispetto alla tradizionale architettura di Von Neumann (istruzioni e dati sullo stesso bus).

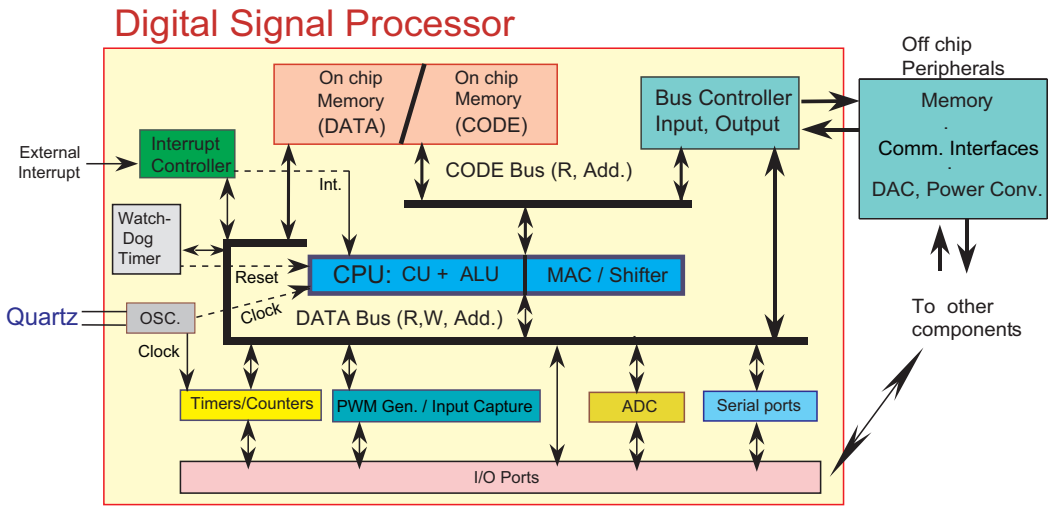


Figura B.2.28: Schema a blocchi di un generico DSP/DSC

Tuttavia, la sola scelta dell'architettura di tipo Harvard non spiega completamente l'orientamento del DSP all'elaborazione di segnali. Per capire la particolarità di questi processori occorre anzitutto considerare quali sono le operazioni necessarie nelle applicazioni che richiedono DSP, le quali si possono ricondurre sostanzialmente alle categorie di **filtraggio** e **convoluzione** (FFT, modulazioni, demodulazioni, ecc.). In entrambi i casi, il processore deve eseguire un grande numero di operazioni aritmetiche di moltiplicazione e somma. Ad esempio, i filtri digitali sono descrivibili tramite funzioni di trasferimento F che legano l'ingresso all'uscita con una formula del tipo $Y = F \cdot U$, se espresse secondo le Z-trasformate[24] con una forma razionale fratta del tipo ($m < n$):

$$F(z) = \frac{N(z)}{D(z)} = \frac{b_0 z^m + b_1 z^{m-1} + \cdots + b_m}{z^n + a_1 z^{n-1} + \cdots + a_n}$$

Più comunemente, la Z-trasformata è scritta nella forma in cui compaiono solo potenze di z^{-1}:

$$F(z^{-1}) = \frac{N(z^{-1})}{D(z^{-1})} = \frac{b_0 z^{-(n-m)} + b_1 z^{-(n-m+1)} + \cdots + b_m z^{-n}}{1 + a_1 z^{-1} + \cdots + a_n z^{-n}}$$

[24]Per una trattazione completa del concetto di Z-trasformata si faccia riferimento ad un testo specifico sull'elaborazione di segnali digitalizzati o sulla progettazione di sistemi di controllo digitali.

Dato che z^{-1} rappresenta l'operatore "ritardo unitario", corrispondente ad un periodo di campionamento, quest'ultima forma può essere facilmente convertita in una equazione alle differenze, cioè l'operazione che viene effettivamente eseguita dal processore sui valori campionati agli istanti $k, k-1, k-2$ ecc., del segnale in ingresso (m valori, $u(k), u(k-1)$, ecc.) e del segnale filtrato in uscita (n valori, $y(k), y(k-1)$, ecc.)[25]:

$$Y(z)D(z^{-1}) = U(z)N(z^{-1})$$

$$\Rightarrow y(k) = -a_1 y(k-1) - \cdots - a_n y(k-n) + b_0 u(k-n+m) + \cdots + b_m u(k-n)$$

Per implementare (ad esempio) in codice C le equazioni precedenti, si potrebbe anzitutto memorizzare i campioni dei segnali u e y e i relativi coefficienti moltiplicativi in degli array di dimensione opportuna, rispettivamente M e N. Poi, considerando gli elementi di indice 0 corrispondenti ai valori associati all'istante k e quelli ad indice superiore corrispondenti ai campioni precedenti (i.e. $k-1$, $k-2$, ecc.), il codice completo potrebbe essere il seguente:

```
float y[N],u[M];   // VALORI CAMPIONATI
float a[N], b[M]; // COEFFICIENTI
int i;            // INDICE per l'aggiornamento dei dati
float T;          // SAMPLE-TIME
...
while(1){
  // LETTURA INGRESSO CORRENTE: istante k = indice 0
  u[0] = read_input();
  // CALCOLO USCITA: istante k = indice 0, istante k-n = indice N
  y[0] = -a[1]*y[1] - ... -a[N]*y[N] + b[0]*u[0] + ... + b[M]*u[M];
  write_output(y[0])
  // AGGIORNAMENTO CAMPIONI (preparazione al ciclo successivo)
  for(i = M; i>0; i--) u[i] = u[i-1];
  for(i = N; i>0; i--) y[i] = y[i-1];
  delay(T);
}
```

Come si vede, se l'ordine del filtro è molto elevato, ad ogni istante di campionamento occorre eseguire un gran numero di moltiplicazioni, sommando nel contempo tutti i risultati. Pertanto, l'obiettivo di progetto di un processore di tipo DSP è massimizzare l'efficienza di esecuzione di queste operazioni, integrando nel componente dei blocchi **moltiplicatori** hardware, affiancati alla tradizionale **ALU**, in modo tale da poter eseguire una moltiplicazione ed una somma ("accumulazione") **in parallelo** nello stesso ciclo macchina, grazie a particolari istruzioni chiamate **Multiply/ACcumulate (MAC)**. Inoltre, considerando la rappresentazione binaria di numeri interi, l'operazione MAC richiede un registro di risultato di dimensione doppia rispetto a quella degli operandi. Ad esempio, il risultato della moltiplicazione tra due numeri interi a 16 bit è a 32 bit. Poichè 16 bit si ipotizza essere anche la dimensione "nominale" delle variabili in Data Memory, per poter trasferire il risultato del MAC in memoria è necessario traslare i 16 bit più significativi di 16 posizione verso i bit meno significativi. Anche tale operazione è così comune nelle elaborazioni dei DSP che può essere effettuata simultaneamente alle operazioni di MAC e sempre nel medesimo ciclo macchina, grazie a moduli hardware chiamati **Barrel Shifter**.

Infine, la terza funzionalità principale per la realizzazione efficiente dei filtri digitali è la gestione della memoria contenente i dati campionati tramite strutture a **buffer circolare**. Si può infatti facilmente osservare, dall'implementazione in C del filtro mostrato in precedenza, che a seguito dell'operazione di filtraggio vera e propria all'istante è necessario predisporre gli array con i dati campionati per il successivo (futuro) passo di campionamento, "spostando" l'elemento in posizione k nella posizione $k-1$, quello in posizione $k-1$ in posizione $k-2$ e così via, arrivando a cancellare (i.e. sovrascrivendolo) gli elementi più vecchi e non più necessari per il filtraggio. Lo spostamento dei dati realizzato con i cicli `for(..)` mostrati richiede però un numero molto elevato di assegnazioni tra elementi dell'array, soprattutto in caso di filtri di ordine elevato. Realizzando la memorizzazione dei dati con la tecnica del cosiddetto *buffer circolare*, invece, lo spostamento dei dati è solamente *virtuale*, in quanto è l'indice corrispondente all'istante di campionamento attuale l'unico dato che deve essere modificato per predisporre l'array al successivo passo di campionamento, come mostrato in Figura B.2.29.

[25] Si noti che l'equazione alle differenze ottenuta è quella di un IIR.

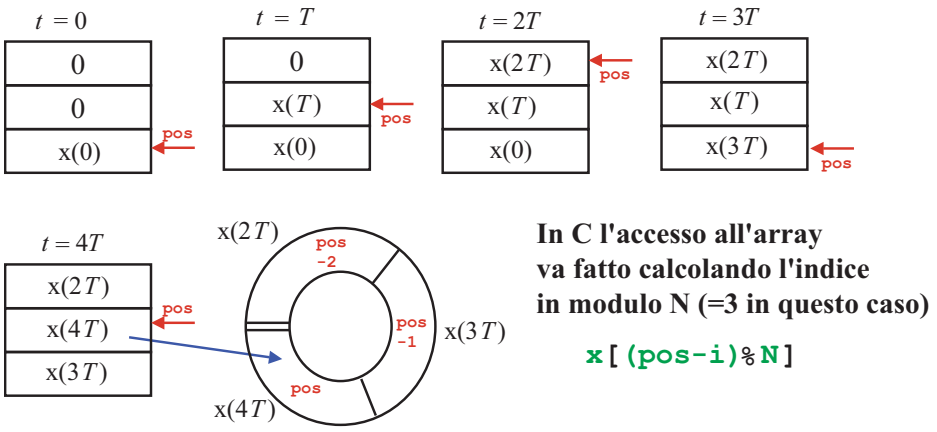


Figura B.2.29: Struttura dati a buffer circolare, per la memorizzazione dei campioni di un segnale

La corretta gestione della struttura dati richiede che l'indice (o del puntatore) venga sempre manipolato aritmeticamente in **modulo** rispetto all'ordine del filtro, cioè alla dimensione utile dell'array. La soluzione *software* a questa problematica è mostrata in Figura B.2.29: la variabile `pos` (compresa sempre fra 0 e 2) è l'indice valido per l'istante attuale, perciò `pos-1` è l'istante precedente; l'operazione di modulo (operatore `%` in C) garantisce che `pos-i` sia sempre compresa tra 0 e 2 e rispetti l'ordine temporale dei campioni. Nei DSP la gestione del buffer circolare è direttamente supportata via **hardware**, tramite la modalità di accesso ai dati denominata **Modulo Addressing**. Ad esempio, nei DSC a 32 bit della Texas Instruments serie C2000 (es. TMS320C28xxx), vi sono 7 registri a 32 bit per l'indirizzamento indiretto dei dati (i.e. *puntatori*), da XAR1 a XAR7. Tra questi, il registro XAR6 può essere usato come puntatore di un buffer circolare, la cui dimensione è definita dal valore numerico dei primi 8 bit del registro XAR1 (dimensione massima del buffer: 256 elementi). Con la sintassi del linguaggio Assembly C2000:

```
1 MAC P,*AR6%++,*XAR7++
```

si esegue una operazione MAC tra un coefficiente costante indirizzato da XAR7 e un dato indirizzato da XAR6 (la notazione AR6 indica in effetti i primi 16 bit di tale indirizzo), che viene incrementato di una unità, in modulo rispetto al valore di XAR1, dopo l'esecuzione dell'istruzione. Il risultato è trasferito nel **Product Register (P)** a 32 bit, il cui contenuto corrisponde a quello dell'accumulatore a 64 bit, shiftato di 32 posizioni. **Il tutto in un singolo ciclo macchina!**

Riassumendo, le caratteristiche principali di un generico DSP/DSC sono:

1. **architettura Harvard**;
2. **unità di calcolo MAC (Multiply/ACcumulate) con Barrel Shifter**;
3. **Modulo Addressing** (per buffer circolari).

B.2.4.1 DSP/DSC nel controllo ad alte prestazioni

Come è facilmente intuibile, la grande efficienza dei DSP/DSC di realizzare algoritmi che implementano funzioni di trasferimento discrete, anche di ordine elevato, risulta molto appetibile anche nel contesto dei sistemi di controllo per realizzare le funzioni di trasferimento tipiche dei controllori. La più nota e versatile formulazione di un regolatore per sistemi in retroazione è detta PID (Proporzionale-Integrale-Derivativo), la cui uscita $y(t)$ dipende dall'errore $e(t)$ fra misura e setpoint, secondo la legge:

$$y(t) = K_p \left(e(t) + \frac{1}{T_i} \int e(t) + T_d \frac{de(t)}{dt} \right)$$

Si noti che in tale formula, detta *classica* o *standard*, il coefficiente del termine Integrale è inversamente proporzionale a una costante di tempo T_i detta *tempo di integrazione*, indicativa del tempo impiegato dal regolatore per annullare l'errore *a regime* (i.e. minore è T_i, maggiore è l'effetto dell'integrazione e più breve è il transitorio di annullamento dell'errore), mentre la costante di tempo T_d è direttamente proporzionale all'effetto *predittivo* introdotto dal termine Derivativo. Sebbene i principi base per la taratura dei parametri del PID *classico* siano ben noti nella pratica ingegneristica, è altrettanto comunue fare riferimento alla formulazione alternativa detta *parallela*, nella quale cioè ogni termine è moltiplicato per un proprio coefficiente indipendente e direttamente proporzionale al "peso" del termine stesso, ottenuta ponendo $K_i = K_p/T_i$ e $K_d = K_p T_d$. Applicando la trasformata di Laplace, le due formulazioni diventano (*classica* a secondo membro, *parallela* al terzo):

$$\mathrm{PID}(s) = K_p \left(1 + \frac{1}{T_i s} + T_d s\right) = K_p + K_i \frac{1}{s} + K_d s$$

La realizzazione digitale di tale funzione di trasferimento a tempo continuo può essere definita anche senza ricorrere, inizialmente, alle Z-trasformate viste in precedenza, considerando che nel dominio a tempo discreto:

- l'integrazione di un segnale nel tempo corrisponde ad una sommatoria dei campioni successivi del segnale errore, moltiplicati per il tempo di campionamento T;
- la derivata dello stesso segnale rispetto al tempo è approssimabile dal rapporto incrementale, cioè la differenza tra due campioni successivi del segnale divisa per il tempo T.

Applicando tali considerazioni:

$$y(k) = K_p \left[e(k) + \frac{1}{T_i} \sum_{i=0}^{k} e(i)T + T_d \frac{e(k) - e(k-1)}{T}\right]$$

Una possibile implementazione in linguaggio C del regolatore PID *classico* è quindi la seguente:

```
float error, error_prev, setpoint, measure; // INPUT
float control_out;        // OUTPUT
float integral = 0;       // ACCUMULATORE INTEGRALE
float Kp, Ti, Td, T;      // COEFFICIENTI
...
while(1){
      error = setpoint - measure;     // Calcolo errore
      integral += error*T;            // Accumulo dell'errore (''integrazione'')
      control_out = Kp*error;                    // Termine P
      control_out += 1/Ti*integral;              // Termine I
      control_out += Td*(error - error_prev)/T; // Termine D
      error_prev = error;             // Memorizzazione errore per il ciclo futuro
      delay(T);                       // Attesa del tempo di campionamento
}
```

Nella pratica, una realizzazione più efficiente del codice di controllo PID richiede certamente alcuni accorgimenti in più, in particolare per filtrare il risultato dell'operazione di derivata e per evitare il cosiddetto *windup* dell'integrale.

Filtraggio della derivata numerica

L'operazione di derivata numerica tende ad amplificare eventuali disturbi nel segnale elaborato. Sebbene la letteratura sul filtraggio di segnali digitali fornisca molte soluzioni (cercando le parole chiave *smoothed differentiation*), per gli scopi di del controllo PID è generalmente sufficiente introdurre un termine passa-basso del primo ordine (con costante di tempo T_f) nella funzione di trasferimento della derivata, che nel dominio tempo continuo diventa quindi:

$$D_{filt}(s) = \frac{s}{T_f s + 1}$$

Tale funzione può essere rielaborata per l'implementazione digitale utilizzando la formula di trasformazione con il metodo delle "differenze all'indietro" (*backward difference*[26]), corrispondente al passaggio che lega la variabile di Laplace s alla variabile z con la sostituzione $s = \frac{z-1}{Tz} = \frac{1-z^{-1}}{T}$, applicando la quale si ottiene la funzione (già espressa rispetto a z^{-1}):

$$D_{filt}(z^{-1}) = \frac{1 - z^{-1}}{(T_f + T) - T_f z^{-1}}$$

Considerando tale funzione come un filtro sul segnale $e(.)$ per ottenere la derivata $d(.)$, con costante T_f in genere posta ad un valore tra 2 e 10 volte il tempo di campionamento T, il calcolo è:

$$d(k) = \frac{T_f d(k-1) + e(k) - e(k-1)}{T + T_f}$$

Logiche di anti-windup dell'integrale

Il problema del *windup* dell'integrale è principalmente legato al fatto che l'uscita del regolatore non può assumere valori arbitrariamente grandi, a causa dei limiti tecnologici dell'attuatore a cui è associata (v. Figura B.2.30).

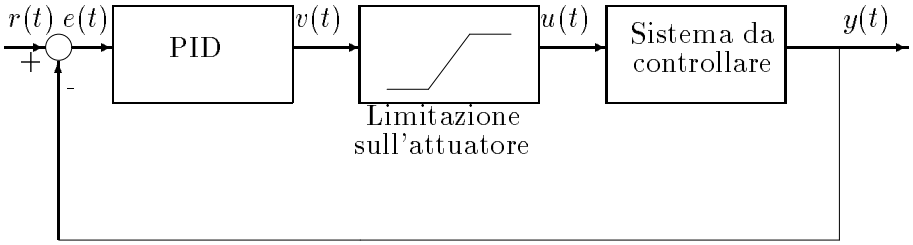


Figura B.2.30: Schema di controllo in retroazione con saturazione dell'attuatore

Qualora la somma dei tre termini P, I e D generi un valore maggiore di tali limiti, il sistema di controllo si trova in condizioni nonlineari (i.e. saturazione del comando all'attuatore) e, di fatto, agisce in catena aperta anzichè ad anello chiuso, in quanto il segnale calcolato dal controllore in base alla retroazione non corrisponde al comando attuato sul sistema da controllare. Normalmente, ciò corrisponde anche ad una risposta più lenta da parte di quest'ultimo, risposta che determina a sua volta una persistenza di valori elevati dell'errore per un lungo periodo di tempo. Durante questo tempo, continuare ad accumulare i campioni dell'errore porterebbe l'integrale ad assumere valori sempre maggiori (*windup*, appunto), con la conseguenza che quando l'errore tenderà finalmente a diminuire il controllore continuerà ad applicare valori troppo alti del segnale di controllo, a causa della "inerzia" introdotta dall'integrale, e si avranno oscillazioni dell'uscita misurata rispetto al setpoint con tempi di assestamento molto lunghi. La Figura B.2.31 riporta l'andamento della velocità di un'automobile (grafici in alto, set-point 20 m/s), regolata tramite *cruise-control* con controllo PID, e del segnale di comando dell'acceleratore (*throttle*, grafici in basso) in una situazione di saturazione (i.e. *throttle* > 1) causata da un improvviso aumento della pendenza stradale.

Nei grafici di sinistra, il controllore è affetto dal windup del termine integrale: nel grafico dell'acceleratore (in basso), la linea tratteggiata rappresenta il valore di uscita del PID, mentre la linea continua è il valore realmente attuato. Nei grafici di destra, l'integratore NON accumula il segnale errore durante il transitorio in saturazione, con conseguente effetto *anti-windup*.

Per ottenere tale effetto, è quindi sufficiente eseguire l'aggiornamento dell'integrale solo se l'uscita del regolatore PID è attuabile senza saturazioni, tenendo conto dei valori massimo e minimo imposti da queste ultime.

[26] https://www.dsprelated.com/freebooks/pasp/Finite_Difference_Approximation.html

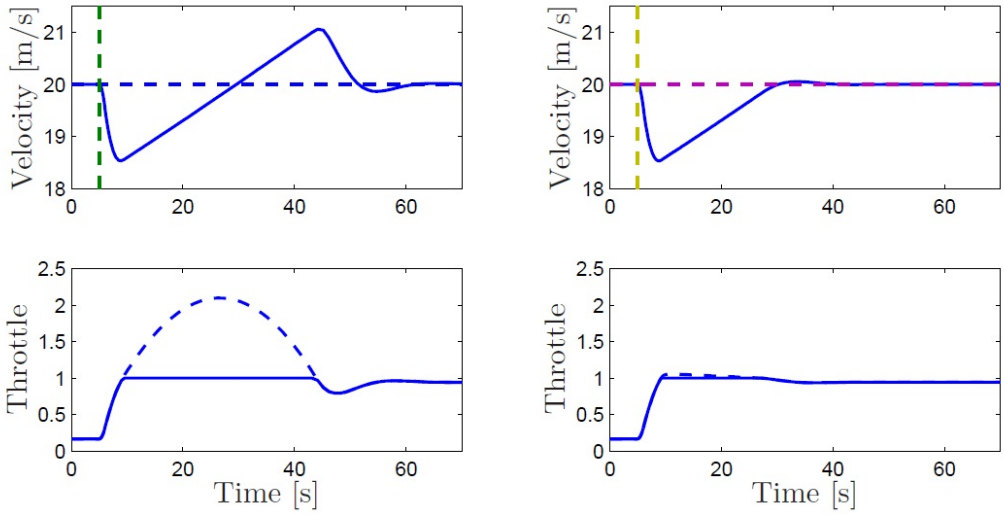


Figura B.2.31: Simulazione del controllo PID di velocità di un automobile (*cruise-control*): a sinistra con windup dell'integrale, a destra con inserimento di un'azione anti-windup. Fonte: "Feedback Systems" di K.J.Åström, R.M. Murray (capitolo 11, "PID Control"): http://www.cds.caltech.edu/~murray/amwiki/index.php/Second_Edition

Codice per PID con anti-windup e derivata filtrata

Le considerazioni descritte in precedenza si possono tradurre in linguaggio C come mostrato nel listato seguente:

```
float error, error_prev, setpoint, measure; // INPUT
float control_out;          // OUTPUT
float derivative = 0;       // DERIVATA
float integral = 0;         // ACCUMULATORE INTEGRALE
float outMax, outMin;       // LIMITI DI SATURAZIONE
bool  saturated;            // FLAG
float Kp, Ti, Td, Tf, T; // COEFFICIENTI
...
while(1){
      error = setpoint - measure;    // Calcolo errore
      // Calcolo derivata (filtraggio tipo IIR)
      derivative = (Tf*derivative + error - error_prev)/(T+Tf);
      if(!saturated)                 // Se NON in saturazione:
          integral += error*T;       //    aggiornamento integrale
      control_out = Kp*error;        // Termine P
      control_out += 1/Ti*integral; // Termine I
      control_out += Td*derivative; // Termine D
      // Verifica dei limiti di saturazione (massimo/minimo)
      if(control_out > outMax){
         control_out = outMax;
         saturated = 1;
      }
      else if(control_out < outMin){
              control_out = outMin;
              saturated = 1;
      } else saturated = 0;
      error_prev = error;          // Memorizzazione errore per ciclo futuro
      delay(T);                    // Attesa del tempo di campionamento
}
```

Implementazione PID ottimizzata per DSP/DSC

Sebbene efficace da un punto di vista controllistico e "leggibile", il codice precedente NON è ottimale per l'implementazione su DSP/DSC. Per sfruttare le peculiarità di questi dispositivi, prima fra tutte la presenza dell'unità MAC, è opportuno ricorrere alle formulazioni tipiche dei filtri digitali basate sulle Z-trasformate, come mostrato all'inizio della Sezione B.2.4. Tornando alla funzione di trasferimento del PID "ideale" (nella forma parallela), se ne può considerare la rielaborazione in forma digitale utilizzando la formula di trasformazione con il metodo delle "differenze all'indietro" (cioè sostuendo $s = \frac{z-1}{Tz}$), ottenendo la seguente funzione Z-trasformata:

$$\mathrm{PID}(z) = K_p + K_i \frac{Tz}{z-1} + K_d \frac{z-1}{Tz}$$

che può esssere a sua volta posta (con gli opportuni valori dei coefficienti di numeratore e denominatore, in termini di K_p, K_i e K_d) ad una forma:

$$\mathrm{PID}(z^{-1}) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{a_0 + a_1 z^{-1}}$$

con i coefficienti (da calcolare solo in fase di inizializzazione del codice del PID):

$$b_0 = K_i T^2 + K_p T + K_d;\ b_1 = -(K_p T + 2K_d);\ b_2 = K_d;\ a_0 = T;\ a_1 = -T.$$

Quest'ultima espressione è immediatamente riconducibile ad una equazione alle differenze assolutamente analoga a quella di un qualunque filtro digitale (e "solo" di ordine 2), ottimale per l'implementazione su DSP/DSC.

Un altro approccio molto seguito per la programmazione di algoritmi PID su DSP/DSC, è basato direttamente sulla riscrittura della formula alle differenze presentata all'inizio della Sezione B.2.4.1, qui riscritta nella forma parallela:

$$y(k) = K_p e(k) + K_i \sum_{i=0}^{k} e(i)T + K_d \frac{e(k) - e(k-1)}{T}$$

Tale formula, corrispondente ad un filtro FIR, si può ricondurre alla cosiddetta forma *incrementale*, corrispondente a un filtro IIR, applicando considerazioni già presentate nella Sezione A.3.3.2 (Parte A di queste dispense), cioè considerando l'espressione equivalente di $y(k-1)$ ed elaborando i termini della differenza $y(k) - y(k-1)$:

$$y(k) = y(k-1) + K_p\left[e(k) - e(k-1)\right] + K_i T e(k) + \frac{K_d}{T}\left[e(k) - 2e(k-1) + e(k-2)\right]$$

Raccogliendo opportunamente i termini per ridurre il numero di operazioni da implementare nel codice eseguito ciclicamente (ad ogni passo di campionamento), si ottiene:

$$y(k) = y(k-1) + A_0 e(k) + A_1 e(k-1) + A_2 e(k-2)$$

con i coefficienti (da calcolare solo in fase di inizializzazione del codice del PID):

$$A_0 = K_p + K_i T + K_d/T;\ A_1 = -(K_p + 2K_d/T);\ A_2 = K_d/T.$$

Si noti, infine, che la problematica di saturazione dell'uscita può essere risolta, ottenendo un effetto *anti-windup*, rielaborando la funzione di trasferimento digitale secondo lo schema di Figura B.2.32, nel quale $N(z)$ è il numeratore della funzione di trasferimento del PID e $D(z)$ è il suo denominatore.

In assenza di saturazione (i.e. in condizioni di linearità), infatti, la funzione di trasferimento è la quella "nominale":

$$PID(z) = N(z)\frac{1}{1-(1-D(z))} = \frac{N(z)}{D(z)}$$

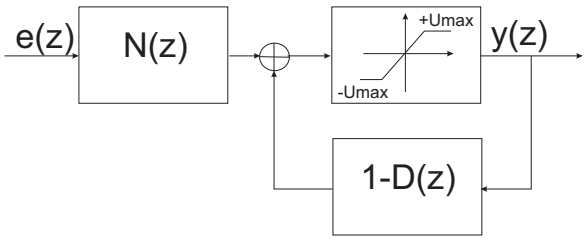


Figura B.2.32: Schema di realizzazione di un regolatore digitale con effetto *anti-windup*

mentre in caso di saturazione i valori memorizzati nell'array dei campioni dell'uscita $y(k)$, $y(k-1)$, ecc., sono tutti limitati, evitando quindi l'effetto *windup*. Questa soluzione, a differenza di quella descritta in precedenza con l'esecuzione condizionata dell'accumulo nella variabile `integral`, è ottimale per l'implementazione su DSP/DSC, in quanto corrisponde di fatto ad imporre una saturazione sul risultato della sequenza di moltiplicazione e somme che risultino al secondo membro in tutte le equazioni alle differenze presentate in questa sotto-sezione. Tale operazione di saturazione del risultato è nativamente supportata via hardware dai moduli MAC di pressochè tutti i DSP/DSC[27].

DSP/DSC negli azionamenti elettrici

Considerando la notevole efficienza computazionale nell'eseguire calcoli come quelli sopra descritti, le risorse computazionali fornite dai DSP/DSC permettono l'implementazione di schemi di controllo anche molto complessi, come quelli richiesti, nei casi di interesse pratico per l'automazione industriale, nel campo degli azionamenti elettrici ad elevate prestazioni (v. Figura B.2.33).

Il processore può effettivamente farsi carico di un numero molto maggiore di elaborazioni, rispetto a quanto potrebbe fare un microcontrollore con frequenza di clock comparabile, eseguendo:

- Filtraggio digitale dei segnali provenienti dai sensori.
- Trasformazioni di coordinate (trifase-bifase, rotore-statore) necessarie per il controllo di coppia nei motori trifase sincroni (Brushless DC e sinusoidale) e asincroni (AC ad induzione).
- Algoritmi di compensazione delle oscillazioni di coppia[28].
- Algoritmi di stima del flusso di induzione (Motori AC Asincroni), e addirittura della posizione (controllo senza sensori o *sensorless*).

Nello stesso tempo impiegato da un microcontrollore per il solo calcolo dell'algoritmo PID, un DSP/DSC può eseguire tutte le elaborazioni sopra citate (v. Figura B.2.34).

La Figura B.2.35 mostra uno schema di progetto con DSP/DSC per la realizzazione di un azionamento elettrico con controllo *sensorless* per motori sincroni a magneti permanenti (**Permanent Magnet Synchronous Motor, PMSM**).

Tra le funzionalità implementate[29] si possono evidenziare:

[27] **NOTA BENE**: l'esecuzione standard di un ciclo di moltiplicazioni e somme su ALU standard, se non saturata via software, potrebbe invece causare un *overflow* del risultato, condizione ben diversa dalla saturazione e deleteria per gli scopi del controllore in retroazione! Come ulteriore approfondimento, si suggerisce di consultare la documentazione delle funzioni PID implementate nella libreria CMSIS-DSP per processori ARM Cortex-Mx, alla pagina https://arm-software.github.io/CMSIS_5/DSP/html/group__PID.html, che mostra come l'implementazione sia analoga alla forma incrementale qui descritta, a meno del coefficiente moltiplicativo T (non considerato esplicitamente dalla libreria CMSIS, ma implicitamente "inglobato" nei coefficienti K_i e K_d), e che suggerisce all'utilizzatore di prestare attenzione al comportamento delle funzioni in caso di overflow.

[28] La coppia erogata da un motore elettrico è tipicamente proporzionale alla corrente fornita dal circuito di alimentazione. La modulazione PWM della tensione di alimentazione genera oscillazioni sulla corrente, pertanto anche sulla coppia erogata.

[29] Per un approfondimento sul controllo sensorless di motori PMSM si consulti l'Application Note 1078 di Microchip: http://ww1.microchip.com/downloads/en/Appnotes/01078B.pdf

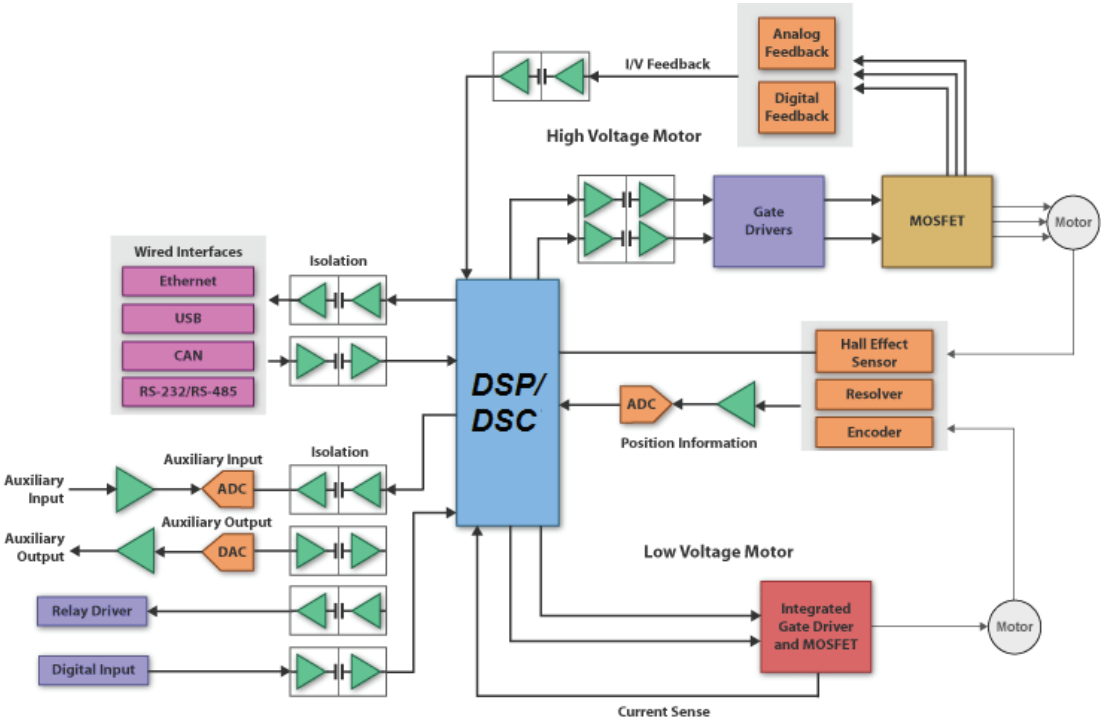


Figura B.2.33: DSP/DSC per il controllo di motori elettrici

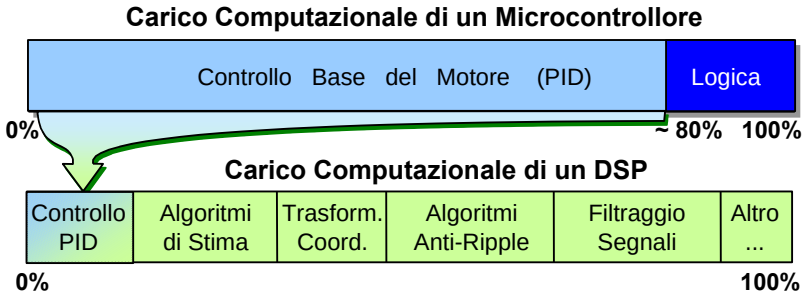


Figura B.2.34: Confronto microcontrollori contro DSP/DSC nelle applicazioni di controllo per azionamenti elettrici

- il controllo ad anello chiuso del convertitore AC/DC per garantire una tensione continua stabile (**Vdc** misurata tramite partitore di tensione collegato ad un ingresso analogico, **ADCIN**) sul convertitore di potenza trifase, gestito tramite il transistor identificato con la sigla VT0 in alto a sinistra nello schema;
- il comando tramite **6 segnali PWM sincronizzati** dei corrispondenti transistor, identificati con le sigle VT1-6, del convertitore trifase (topologia *inverter*) che regola le correnti nelle fasi del motore;
- le trasformazioni dei valori di tensioni/correnti da rappresentazione trifase a bifase (**trasformata di Clark**) e da coordinate di rotore a quelle di statore (**trasformata di Park**), operazioni necessarie per il cosiddetto controllo a **orientamento di campo (Field Oriented Control, FOC)**, che massimizza l'efficienza di generazione della coppia motrice;

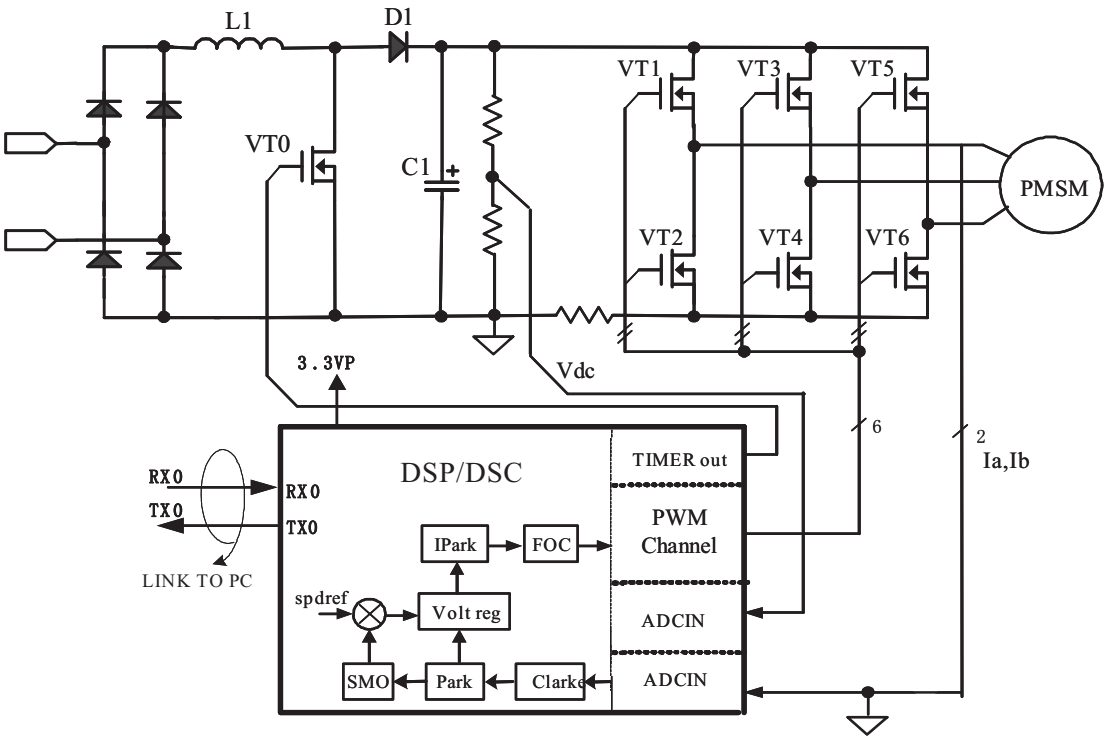


Figura B.2.35: Controllo *sensorless* di motori elettrici sincroni a magneti permanenti (PMSM: Permanent Magnet Synchronous Motor) con DSP/DSC

- l'osservatore di tipo Sliding Mode o **Sliding-Mode Observer (SMO)**, per la stima di posizione (necessaria per le trasformate di Park e Park inversa) e velocità di rotore (controllata in retroazione dallo schema complessivo);

B.2.4.2 *Speeding up*: Fixed-point contro Floating-point

L'elevata ottimizzazione nei DSP delle operazioni di filtraggio e controllo digitale, ha anche un rovescio della medaglia. Prima di tutto, la realizzazione dei moltiplicatori hardware è molto più complessa nel caso in cui il processore debba supportare anche operazioni su valori numerici rappresentati con il formato **virgola mobile (o Floating-Point)**, cioè con la notazione *mantissa-esponente* memorizzati con due sequenze di bit separate. Di conseguenza, i DSP con capacità Floating-Point sono molto più costosi ed il loro utilizzo è riservato ad applicazioni che richiedono una elevata risoluzione numerica, oppure possono "pretendere" un costo maggiore per unità prodotta (es. sofisticati e costosi sistemi di visione artificiale) o abbattere i costi grazie all'economia di scala (es. integrazione in prodotti elettronici di largo consumo, come smartphone). I DSP che supportano solamente calcoli di *aritmetica intera*, cioè con valori numerici rappresentati in **virgola fissa (Fixed-Point)**, sono invece maggiormente diffusi nelle applicazioni prettamente controllistiche ed offrono un rapporto costo/prestazioni elevato, seppur bilanciato dalla maggiore complessità di programmazione, come verrà discusso nel seguito.

Inoltre, occorre notare che le operazioni di **divisione** risultano di particolare complessità per un processore ottimizzato per la moltiplicazione. Molti DSP/DSC (sia Fixed-Point che Floating-Point) NON includono nel proprio linguaggio Assembly una istruzione di divisione vera e propria o, almeno, non eseguibile in un singolo ciclo macchina. Ad esempio, i DSC Texas Instruments serie C2000 supportano la divisione tra interi senza segno tramite le istruzioni di `conditional subtract`, `SUBCU` (a 16 bit) e `SUBCUL` (a 32 bit), che però devono essere **esplicitamente ripetute** per un

numero di cicli macchina pari alla dimensione in bit degli operandi (es. operandi a 16 bit, 16 cicli), come mostrato dal seguente listato Assembly C2000:

```
; Calculate unsigned: Quot16 = Num16/Den16, Rem16 = Num16%Den16
MOVU ACC,@Num16   ; ACCU = AH : AL, AL = Num16, AH = 0
RPT #15           ; Repeat operation 16 times
SUBCU ACC,@Den16  ; Conditional subtract with Den16
MOV @Rem16,AH     ; Store remainder in Rem16
MOV @Quot16,AL    ; Store quotient in Quot16
```

Per migliorare l'efficienza di elaborazione su DSP/DSC, occorre quindi realizzare programmi nei quali si faccia un limitato uso di operazione di divisione. Possibili accorgimenti da applicare sono:

1. qualora il divisore sia una costante, programmare l'algoritmo in modo che il dividendo sia moltiplicato per il reciproco del divisore.

2. qualora sia possibile approssimare il divisore ad un valore pari ad una potenza di due (es. $2000 \approx 2^{11} = 2048$), programmare una operazione bit-shift del corrispondente esponente, anzichè l'effettiva operazione di divisione. In linguaggio C ciò corrisponde a scrivere:

   ```
   quoziente = dividendo >> 11;
   ```

   anzichè

   ```
   quoziente = dividendo / 2048;
   ```

Per approndire i dettagli sulle elaborazioni Fixed-Point, fondamentali nella programmazione di DSP/DSC e utili per massimizzare l'efficienza computazionale anche su microprocessori e microcontrollori standard, verranno presentati nel seguito alcuni richiami specifici. Come noto, le elaborazioni aritmetiche eseguiti dai calcolatori elettronici si basano su rappresentazioni interne dei valori numerici basate su stringhe di bit. Tuttavia, il significato di una stringa di bit elaborata dal calcolatore è molto differente a seconda della "interpretazione" della stringa stessa. Ad esempio, nel caso più semplice, si può ipotizzare che una stringa di 8 bit rappresenti un numero intero, il cui valore effettivo è legato semplicemente alla conversione fra rappresentazione binaria e rappresentazione decimale, cioè, dato il valore (0 o 1) di ogni bit (b_7 = MSB, b_0 = LSB):

$$b_7 \times 2^7 + b_6 \times 2^6 + \cdots + b_0 \times 2^0$$

In questo modo, l'intervallo di numeri interi che è possibile rappresentare va da 0 a $2^8 - 1 = 255$ (in generale, con N bit: $[0, 2^N - 1]$). Chiaramente, con questa interpretazione non è possibile rappresentare valori negativi, con evidenti limitazioni nell'esecuzione di operazioni di sottrazione. Occorre quindi introdurre la possibilità di rappresentare numeri positivi e negativi, ad esempio assegnando al bit più significativo il ruolo di segno: se 0, segno +, se 1, segno −. L'intervallo di valori rappresentati diventa così $[-2^{N-1}, +2^{N-1}]$, che però comprende una doppia rappresentazione possibile per lo 0 (100..00 e 000..00). Solitamente, la rappresentazione binaria di numeri interi con segno viene effettuata secondo il formato cosiddetto **in complemento a due**, con il quale i valori negativi sono rappresentati invertendo ogni bit della rappresentazione del corrispondente valore assoluto e sommando al risultato il valore 1 (in pratica, dato x valore assoluto, $x_{neg} = 2^N - x$). Con quest'ultimo formato, i valori possibili sono l'intervallo $[-2^{N-1}, 2^{N-1} - 1]$, con un'unica rappresentazione dello 0.

Per quanto riguarda la rappresentazione dei numeri reali, una ipotesi molto efficace potrebbe essere quella di utilizzare una notazione "scientifica", del tipo **Mantissa-Esponente**, riservando quindi determinati bit della codifica per rappresentare il valore della mantissa ed i rimanenti per il valore dell'esponente. Pertanto, ipotizzando che M sia il valore (decimale, positivo o negativo) degli m bit della mantissa, ed E sia il valore (decimale, positivo o negativo) degli e bit dell'esponente, il valore reale rappresentato sarà:

$$M \times 2^E$$

Ad esempio, con una word di 16 bit, dei quali 12 bit rappresentano la mantissa e 4 l'esponente, il più grande numero rappresentabile in complemento a due è $(2^{11} - 1) \times 2^7 = 262216$, mentre il più piccolo è $1 \times 2^{-8} = 0.00390625$ (in valore assoluto).

Quest'ultima modalità è appunto chiamata rappresentazione **Floating-Point**, in quanto durante l'elaborazione è possibile variare molto facilmente il valore dell'esponente e, di conseguenza, la posizione rispetto al punto decimale delle cifre rappresentate dalla mantissa. Purtroppo, tale rappresentazione implica, in genere, una notevole complessità di gestione da parte del calcolatore, che deve effettuare separatamente le operazioni aritmetiche su mantissa ed esponente, qualora non sia dotato di uno specifico modulo hardware ausiliario per i calcoli Floating-Point (i.e. la **Floating-Point Unit, FPU,** spesso integrata nei microprocessori più performanti). Nel caso di processori con architettura *special-purpose*, come i DSP, questa complessità implica anche un notevole incremento del costo di progettazione.

Nella pratica, è comunque possibile rappresentare numeri reali anche solo con il supporto di elaborazione per numeri interi, ipotizzando che il numero rappresentato dalla stringa di bit sia la mantissa di un numero reale, e che l'esponente sia "implicito", cioè determinato dal programmatore in fase di scrittura del software, ma non memorizzato in una stringa di bit del calcolatore. Il valore di questo "esponente virtuale" può essere considerato come la posizione di un punto decimale, che separa i bit della rappresentazione in una parte intera e una parte frazionaria. Ad esempio, la stessa rappresentazione binaria:

$$M = 01100100$$

corrispondente ad un valore decimale (intero) di 100, può essere considerata come un valore reale rappresentandone la parte frazionaria con due bit, ottenendo un valore pari a $M \times 2^{-2}$:

$$011001(.)00 = 25$$

oppure come un valore reale con 7 bit di parte frazionaria, pari a $M \times 2^{-7}$:

$$0(.)1100100 = 0.78125$$

Questa modalità di rappresentazione dei numeri reali viene chiamata **Fixed-Point**. Il formato di un numero Fixed-Point viene solitamente identificato con la sigla $\mathbf{Q}\boldsymbol{N}$, dove N indica il numero di bit della parte frazionario e la dimensione totale della stringa di bit è implicita, o $\mathbf{Q}\boldsymbol{M.N}$, dove M ed N sono rispettivamente il numero di bit della parte intera e il numero di bit della parte frazionaria. Nei due casi precedenti, il primo numero reale è in formato Q2 o Q5.2 (considerando il MSB come bit di segno), ed il secondo in formato Q7. Si noti, che N è anche pari all'esponente (negativo) della potenza di due per la quale bisogna moltiplicare il valore intero, al fine di ottenere il corrispondente valore reale.

Chiaramente, la notazione Fixed-Point richiede un maggiore sforzo in fase di programmazione, in quanto occorre valutare attentamente la compatibilità fra gli operandi (in relazione alla "posizione" del punto decimale) delle varie istruzioni aritmetiche. In particolare:

- per **cambiare la rappresentazione** da QP a QR, occorre shiftare la stringa di bit di P - R posizioni (cioè verso sinistra se R > P, verso destra se P > R). Ad esempio, la conversione di un numero da Q10 a Q12 in linguaggio C corrisponde a:

```
var_q12 = var_q10 << 2;
```

  mentre la conversione opposta è:

```
var_q10 = var_q12 >> 2;
```

  NOTA BENE: qualora si voglia programmare in linguaggio C un microcontrollore o DSP/DSC è bene accertarsi che le operazioni di shift su numeri con segno (*shift aritmetico*) siano implementate (i.e. tradotte in Assembly) con la modalità **sign-extended**, in modo da preservare il bit di segno della rappresentazione in complemento a due. Poichè lo standard ANSI C prevede che il comportamento delle operazioni di shift verso destra sia *implementation-dependent* (cioè possono indifferentemente porre a zero o a uno i bit più significativi nel risultato, a seconda del compilatore in uso), questo dettaglio NON è così scontato.

- **addizioni e sottrazioni** possono essere effettuate solo se gli operandi sono entrambi nello stesso formato QN. Infatti:

$$X \times 2^{-N} + Y \times 2^{-N} = (X + Y) \times 2^{-N}$$

- la **moltiplicazione** fra due numeri qualunque può essere sempre effettuata, tenendo però presente che il prodotto fra un numero QP e un numero QR, dà luogo ad un risultato Q*(P+R)*. Infatti:

$$X \times 2^{-P} \times Y \times 2^{-R} = X \times Y \times 2^{-(P+R)}$$

  In realtà, occorre notare che, nel caso di rappresentazioni binarie in complemento a due, il prodotto fra interi produce un risultato nel quale vi sono **due bit di segno**, uno dei quali deve essere eliminato shiftando il risultato verso sinistra. Pertanto, se X e Y sono in complemento a due, la formula corretta è QP × QR = Q*(P+R+1)*. Si noti, inoltre, che i processori come i DSP, hanno quasi sempre un accumulatore di dimensione doppia rispetto alla dimensione degli operandi ammessi dal moltiplicatore hardware, proprio per fare in modo che il risultato della moltiplicazione possa essere memorizzato con la massima precisione possibile.

- la **divisione** fra interi fornisce un risultato utile solamente se il dividendo è maggiore del divisore, altrimenti il risultato è nullo. Inoltre, il risultato di una operazione QP/QR è un numero Q*(P-R)*. Pertanto, occorre avere l'accortezza di cambiare opportunamente la rappresentazione del dividendo, ovvero di moltiplicarlo per una costante opportuna di **messa in scala**, prima della divisione e poi effettuare l'operazione inversa sul risultato (messa in scala pre- e post-divisione). La massima precisione dell'operazione si ottiene applicando un ragionamento opposto a quello visto prima per la moltiplicazione: per dividere tra loro due numeri QN, è bene scalare il dividendo con rappresentazione Q*(2×N)*. Ad esempio, il modo corretto di esprimere una divisione fra numeri a 16 bit con rappresentazione Q*15* in linguaggio C è il seguente:

```
#include <stdint.h> // Per i tipi int8_t, int16_t ecc.
int16_t a;
int16_t b;
int32_t quot;

if ( b != 0 )
    quot = (((int32_t)a << 15) / ((int32_t)b));
else
    quot = 0;
```

  Si noti che il risultato finale `quot` è un numero Q15, ma a 32 bit (cioè Q16.15).

B.2.4.3 Messa in scala (*scaling*): tecnologica e aritmetica

Le osservazioni descritte in precedenza sull'elaborazioni aritmetiche in notazione Fixed-Point sono particolarmente importanti nella realizzazione di algoritmi di controllo e di filtraggio di segnali, qualsiasi sia il sistema di elaborazione sul quale vengano implementati. Infatti, gli algoritmi di controllo sono in genere progettati considerando il sistema dal punto di vista fisico, pertanto le variabili come l'uscita del sistema (il valore di misura) ed il suo ingresso (l'uscita del controllore) sono espresse nelle unità di misura ingegneristiche relative alle grandezze fisiche in esame (tensioni, velocità di rotazione, correnti, ecc.). La loro realizzazione su calcolatore, richiede invece che le grandezze siano espresse in una rappresentazione opportuna, derivante dalla conversione digitale delle grandezze fisiche. Pertanto, gli algoritmi di controllo necessitano di operazioni di **messa in scala o *scaling*** delle variabili e dei coefficienti del regolatore, in relazione alle costanti di conversione delle grandezze nelle varie rappresentazioni ed unità di misura. Questa operazione di *scaling* in funzione delle unità ingegneristiche è come detto **sempre necessaria** nei sistemi di controllo digitali e viene appunto definita **messa in scala tecnologica**.

Ad esempio, si consideri lo schema di controllo in retroazione, realizzato con un elaboratore digitale, mostrato in Figura B.2.36.

Evidentemente, le grandezze rappresentate all'interno del calcolatore, il set-point Y_{spq}, la misura Y_{mq}, l'errore E_q e l'uscita del regolatore U_q, sono legate alle effettive grandezze fisiche da una relazione

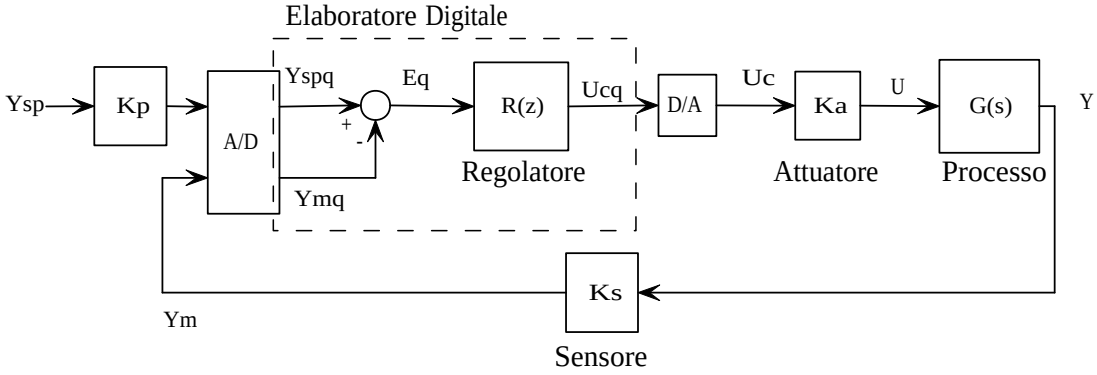


Figura B.2.36: Schema a blocchi per un controllore digitale che evidenzia le costanti per la **messa in scala tecnologica**

proporzionale, dipendente dalle costanti (fattori di scala) dei convertitori A/D e D/A (indicate ad esempio con $K_{a/d}$ e $K_{d/a}$), del sensore e dell'attuatore (indicate con K_s e K_a, nell'ipotesi che siano caratterizzati da relazioni ingresso/uscita lineari). In generale, tali relazioni saranno:

$$Y_{spq} = K_p K_{a/d} Y_{sp} = K_{sp} Y_{sp}$$

$$Y_{mq} = K_s K_{a/d} Y = K_{in} Y$$

$$U = K_{d/a} K_a U_{cq} = K_{out} U_{cq}$$

Comunemente, si suppone che l'algoritmo di controllo sia progettato per elaborare i valori delle grandezze fisiche, in particolare per fornire il valore di controllo U al processo, misurandone l'uscita Y. Ad esempio, un semplice algoritmo di controllo lineare (tipo PI), realizzato in forma discretizzata, potrebbe essere:

$$E(k) = Y_{sp}(k) - Y(k)$$

$$U(k) = aU(k-1) + bE(k)$$

Si noti, innanzitutto, che la prima operazione (calcolo dell'errore) deve essere effettuata su valori dimensionalmente consistenti. Nel caso pratico, se K_{sp} e K_{in} sono diverse, occorre introdurre una costante di messa in scala K_1 opportuna, per rendere Y_{spq} ed Y_{mq} consistenti fra loro (**messa in scala delle variabili d'ingresso**):

$$K_{sp} = K_1 K_{in}$$

Dopodichè occorre effettuare la **messa in scala del controllore** tramite **una** delle due tecniche seguenti:

1. **messa in scala delle variabili** e mantenimento dell'algoritmo originale; in questo caso, occorre riportare i valori delle grandezze provenienti dal convertitore A/D ai valori delle grandezze fisiche, e riportare il valore fornito al converitore D/A al valore effettivamente fornito al processo:

$$E(k) = \frac{1}{K_{sp}} Y_{spq} - \frac{1}{K_{in} K_1} Y_{mq}$$

$$U(k) = aU(k-1) + bE(k)$$

$$U_{cq}(k) = \frac{1}{K_{out}} U(k)$$

2. **messa in scala dell'algoritmo**, cioè dei suoi coefficienti; in questo caso (ipotizzando per semplicità che sia $K_{in} = K_{sp}$):

$$E_q(k) = K_{in} E(k)$$

$$K_{out}U_{cq}(k) = aK_{out}U_{cq}(k-1) + \frac{b}{K_{in}}E_q(k)$$

$$\Rightarrow U_{cq}(k) = aU_{cq}(k-1) + \frac{b}{K_{in}K_{out}}E_q(k)$$

Per realizzare un controllore come quello dell'esempio precedente in aritmetica Fixed-Point, le operazioni di messa in scala richiedono ulteriori accorgimenti. In particolare, è necessario che tutti i coefficienti e le costanti di messa in scala *tecnologica* siano opportunamente rappresentate da variabili "intere", con l'opportuna precisione di rappresentazione. In particolare, è necessario riportare i valori frazionari minori di 1 (es. $1/K_{in}$) a valori interi, evitare l'overflow dei risultati del prodotto (tramite accumulatori o variabili ausiliarie per memorizzare i risultati intermedi delle moltiplicazioni a $2 \times N$ bit, se gli operandi considerati sono a N bit) e, soprattutto, è necessario effettuare eventuali cambiamenti del formato QN dei valori che devono essere sommati fra di loro, qualora la scelta del formato di rappresentazione Fixed-Point non sia omogeneo per tutte le quantità consideate. Tutte queste operazioni costiuiscono la **messa in scala aritmetica** del controllore.

Infine, un ulteriore accorgimento in grado di migliorare notevolmente l'efficienza delle operazioni di messa in scala è quello di fare in modo che tutte le costanti di conversione siano potenze di due. In questo modo, moltiplicazioni e divisioni si trasformano in semplici (e più veloci!) istruzioni di bit-shift. Ad esempio, il progettista può intervenire sulla costante relativa alla conversione A/D (o D/A) considerando che questa dipende dalla risoluzione (N bit) del convertitore e dal range ammissibile per la grandezza in ingresso. Nel caso della misura proveniente dal sensore, che trasforma la grandezza fisica in una tensione di valore massimo V_{max}[30]:

$$K_{a/d} = \frac{2^N}{V_{max}}$$

$$K_{in} = K_s K_{a/d} = \frac{V_{max}}{Y_{max}} \frac{2^N}{V_{max}} = \frac{2^N}{Y_{max}}$$

A questo punto, è opportuno "scegliere" (se possibile) un valore di Y_{max} che sia pari ad una potenza di due, che sarà pertanto la potenza di due più vicina all'effettivo valore massimo ammissibile dell'uscita del sistema (es. $2048 = 2^{11}$ anzichè 2000).

Si consideri, come esempio, di dover implementare un controllore digitale, **con processore ad 8 bit**, per la velocità di un motore elettrico, supponendo che il motore sia dotato di dinamo tachimetrica, che possa raggiungere la velocità massima di 2000 giri/minuto (RPM), che la dinamo fornisca un valore di uscita massimo (a 2000 RPM) di 0,5 V e che tale segnale analogico debba essere acquisito da un convertitore A/D con fondo scala a 5 V. Per utilizzare tutto il range di tensione in ingresso al convertitore A/D, è necessario interporre fra dinamo e ADC un amplificatore con guadagno 10. Approssimando invece il fondo scala della velocità meccanica a 2048 RPM (i.e. 2^{11}) anzichè 2000, la costante di conversione K_{in} diventa pari a:

$$K_{in} = K_{dinamo}\, G_{ampli}\, K_{a/d} = \frac{0,5[V]}{2000[RPM]}\, 10\, \frac{2^N}{5[V]} = \frac{2^N}{2000[RPM]} \approx \frac{2^N}{2048[RPM]} = 2^{N-11}$$

Moltiplicare un valore numerico per tale costante significa, nel dominio digitale, shiftare di $N-11$ bit **verso sinistra** la stringa di bit corrispondente, mentre dividere per tale costante significa shiftare di $N-11$ bit **verso destra** (operazioni di shift che devono essere fatte con mantenimento del segno in caso di valori negativi..). Analoghe considerazioni potranno essere fatte sulla costante K_{out}, note le caratteristiche del convertitore D/A e dell'azionamento di potenza per il motore.

Come esempio (semplificato) di scelta del formato Fixed-Point delle variabili in gioco, si supponga di voler realizzare un algoritmo di controllo come quello descritto in precedenza, utilizzando la messa in scala dell'algoritmo (metodo 2):

$$U_{cq}(k) = aU_{cq}(k-1) + \frac{b}{K_{in}K_{out}}E_q(k) = aU_{cq}(k-1) + b'E_q(k)$$

[30] Si suppone anche, per semplicità, che le grandezze siano tutte variabili in intervalli di valori positivi, tra 0 ed un valore massimo.

Si ipotizzi poi di aver progettato i coefficienti del controllore con i seguenti valori numerici:

$$a = 0,87 \quad b' = 0,0251$$

In questo caso, la scelta più idonea per la loro rappresentazione numerica in notazione Fixed-Point è la seguente:

$$a_{scaled} = 222,72 \approx 222 = (0.)11011110[\text{Q-8}] \quad \Longrightarrow a_{approx} = 0,8671875$$

$$b'_{scaled} = 6,4 \approx 6 = (0.)00000110[\text{Q-8}] \quad \Longrightarrow b'_{approx} = 0,0234375$$

Come si vede, i numeri reali sono stati approssimati in modo che il maggiore dei due (a) possa essere rappresentato con il maggior valore rappresentabile da un numero intero ad 8 bit, valore ottenuto moltiplicando il numero reale di partenza (i.e. 0.87) per un fattore 2^N tale che il risultato non superi 256. Si è inoltre scelto di usare la stessa messa in scala anche per b, nonostante questo sia molto più piccolo di a, al fine di mantenere la consistenza tra le notazioni di tutti i numeri in gioco, piuttosto che ottimizzare la rappresentazione di ciascuno e dover poi omogeneizzare le somme con operazioni di scaling aritmetico. In tal modo, si può scrivere:

$$U_{cq,scaled}(k) = a_{scaled}U_{cq}(k-1) + b'_{scaled}E_q(k) = a\,2^8\,U_{cq}(k-1) + b'\,2^8\,E_q(k)$$

$$\Longrightarrow U_{cq} = U_{cq,scaled}\,2^{-8}$$

Pertanto, il risultato finale dell'algoritmo di controllo dovrà essere shiftato verso destra di 8 prima di poterlo applicare al convertitore D/A. In realtà, ipotizzando che il processore ad 8 bit utilizzi un accumulatore a 16 bit per eseguire moltiplicazioni e somme, sarà sufficiente usare il byte più significativo di questo accumulatore.

Esempio pratico di scaling tecnologico/aritmetico (con offset)

In molti casi i segnali acquisiti da sensori/trasduttori per grandezze fisiche *bipolari* (i.e. con segno) sono riportati a valori di tensione *unipolari* (es. 0÷5 V). L'operazione completa di scaling deve ovviamente tenere conto di questa traslazione dei valori negativi verso un range positivo, corrispondente all'introduzione di un *offset*. Si consideri ad esempio un sensore di corrente di tipo LEM HSLR 10-P, la cui caratteristica di uscita a fronte di una corrente da misurare di entità I_p è la seguente:

$$V_{out} = I_p * 0,046 + V_{ref}$$

In questo caso, la costante di proporzionalità del sensore K_s, anche indicata come *sensitivity* nei datasheet di sensori/trasduttori commerciali (v. Sezione A.1.4.1 della Parte A di queste dispense), è pari a 0,046 $[A/V]$. La tensione V_{ref} è un offset configurabile, la cui scelta più consigliabile è normalmente pari alla metà del range di tensione ammissibile dal convertitore A/D utilizzato per acquisire il dato dal sensore. Ad esempio, supponendo di utilizzare un convertitore A/D a 10 bit con ingresso compreso tra 0 e 5 V, si porrà V_{ref} = 2,5 V. L'operazione di messa in scala tecnologica completa risulta quindi:

$$I_{acq} = K_sK_{a/d}I_p + K_{a/d}V_{ref} = 0,046\frac{2^{10}}{5}I_p + \frac{2^{10}}{5}V_{ref}$$

Il valore I_{acq} corrisponde al numero intero fornito in uscita dal convertitore A/D e la costante di messa in scala $K_{in} = K_sK_{a/d}$ è quindi pari a 9,4208. Utilizzando aritmetica Fixed-Point, per riportare calcoli progettati sulla base della grandezza fisica originale (i.e. corrente I_p espressa in A) alla corrispondente quantità scalata, si deve invertire la formula di scaling tecnologico ed esprimere tutte le costanti moltiplicative con numeri interi:

$$I_{q,p} = (I_{acq} - K_{a/d}V_{ref}) * K_Q * \frac{1}{K_{in}}$$

dove K_Q è il fattore necessario a trasformare la quantità $1/K_{in}$ in un numero intero con rappresentazione QN opportuna. Ad esempio, volendo utilizzare una rappresentazione Q12 si avrà K_Q = 4096 e quindi ($K_Q * [1/K_{in}] = 434,7826$):

$$I_{q,p} = (I_{acq} - 512) * 434$$

NOTA BENE: Laddove non sia esplicitamente indicata dal costruttore come *sensitivity*, K_s può essere ricavata dal rapporto tra l'ampiezza dell'intervallo del segnale elettrico in uscita e quella dell'intervallo della grandezza fisica misurata, come appunto riportato nella Sezione A.1.4.1 di queste dispense. Ad esempio, si consideri un sensore di pressione la cui uscita sia una tensione compresa tra 1 e 5 V, in relazione ad una pressione variabile tra -20 bar e +20 bar. In tal caso:

$$K_s = \frac{(5-1)[V]]}{[+20-(-20)][bar]} = \frac{4}{40}[V/bar] = 0,1[V/bar]$$

Inoltre, il sensore presenta un offset (da considerare come nel caso del sensore LEM HSLR 10-P già descritto) pari alla tensione di uscita corrispondente al valore nullo della pressione. Poichè tale valore nullo corrisponde anche al valor medio tra i due estremi dell'intervallo di pressione misurabile, l'offset sull'uscita è anch'esso il valor medio dell'intervallo $1 \div 5$ V, quindi pari a 3 V.

B.2.5 Il DSC Microchip dsPIC33FJ128MC802

Un ulteriore esempio di processori orientati al controllo, con caratteristiche da DSC ma versatilità e facilità d'uso da microcontrollore, sono quelli della famiglie dsPIC30 e dsPIC33 di Microchip. Questi ultimi, di sviluppo più recente, sono caratterizzati da tensioni di esercizio più basse (alimentazione a 3.3 V anzichè 5 V), prestazioni migliori e funzionalità più estese. Nel seguito verrà descritto più in dettaglio il dsPIC33FJ128MC802, considerato un valido strumento per attività di laboratorio nella didattica universitaria, ma utilizzabile anche per applicazioni industriali e sofisticate di controllo automatico.

Le caratteristiche principali del dsPIC33FJ128MC802 sono riassunte in Figura B.2.37, dalla quale si può evidenziare:

- CPU con funzioni DSP (*DSP Engine*) su operandi a 16 bit con accumulatori a 40 bit, architettura Harvard e capacità di calcolo fino a 40 MIPS.
- Timer/Counter sia a 16 che 32 bit.
- Modulatori PWM orientati al controllo di motori elettrici (6 canali accoppiati a due a due per pilotare convertitori trifase).
- Moduli per Input Capture (misure di frequenza) e Output Compare (altri PWM, ma indipendenti e non sincronizzati).
- Contatori specifici per encoder incrementali (QEI: Quadrature Encoder Interface).
- Convertitori A/D con risoluzione configurabile (10 o 12 bit).
- Numerose porte di comunicazione (seriali sincrone e asincrone, protocollo CAN).
- Modulo DMA (Direct Memory Access), per velocizzare il trasferimento di dati dalla RAM alle periferiche integrate e viceversa (es. scansione degli ingressi analogici e copiatura dei risultati in un array di memoria, interamente gestita via hardware).

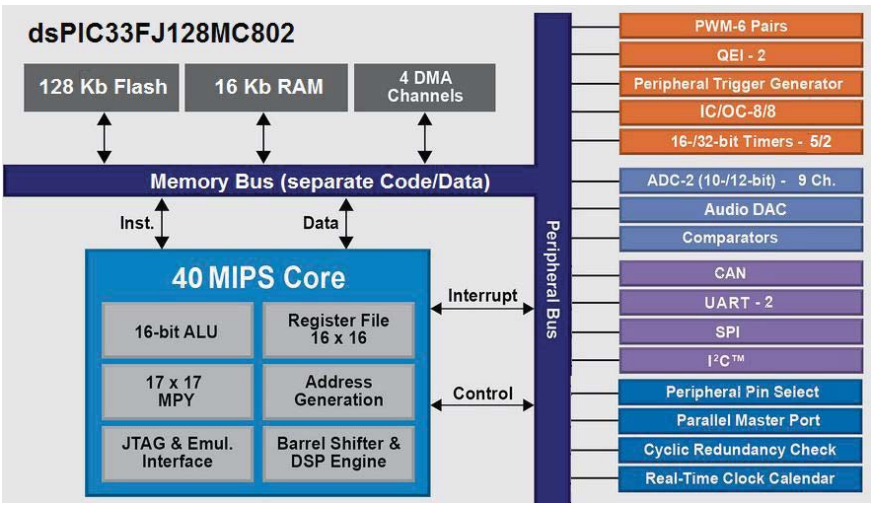


Figura B.2.37: Schema sintetico del dsPIC33FJ128MC802

B.2.5.1 Funzionalità DSP

Il *DSP Engine* del processore dsPIC33 è caratterizzato dalla seguente modalità operativa, descritta con riferimento alla funzionalità principale di *Multiply/ACcumulate* (MAC):

1. la CPU utilizza complessivamente 16 registri dato a 16 bit, da **W0** a **W15**; tra questi, da W4 a W7 sono riservati come operandi di istruzioni DSP, mentre quelli da W8 a W11 come puntatori per l'indirizzamento indiretto in istruzioni DSP;

2. gli operandi (a 16 bit) per le istruzioni DSP sono prelevati simultaneamente grazie a due distinti bus di accesso alla RAM: **X Data Bus** (read/write) e **Y Data Bus** (read-only);

3. il modulo moltiplicatore esegue l'operazione richiesta tra `MAC`, `MSC` (*Multiply/SubtraCt*), `MPY` (*MultiPlY*) o `MPY.N` (*MultiPlY-and-Negate*) e trasferisce il risultato, tramite il **Barrel Shifter** con estensione del segno, in uno tra due accumulatori a 40 bit;

4. grazie alla dimensione degli accumulatori è possibile eseguire più moltiplicazioni in sequenza riducendo le possibilità di overflow del risultato; tale condizione può comunque essere evitata abilitando una logica di saturazione degli accumulatori, rispetto al limite rappresentabile con 39 bit (più bit di segno) oppure ad un limite più conservativo a 31 bit (più bit di segno);

5. i 40 bit contenuti negli accumulatori possono essere trasferiti in RAM tramite X Data Bus, operazione che scarta forzatamente i 16 bit meno significativi e gli 8 più significativi.

L'esempio più completo di istruzione DSP è il seguente:

```
1 MAC W4*W6, A, [W8]+=2, W4, [W10]+=2, W6, [W13]+=2
```

la quale compie le operazioni nell'ordine, ma **nello stesso ciclo macchina**:

1. moltiplica il contenuto dei registri `W4` e `W6`, con risultato nell'accumulatore `A`;

2. trasferisce una nuova coppia di operandi, rispettivamente dalla locazione di X Data Memory puntata da `W8` e dalla locazione di Y Data Memory puntata da `W10`;

3. incrementa di due (in modo da mantenere l'allineamento ai 16 bit) gli indirizzi in W8 e W10;

4. trasferisce il contenuto dell'accumulatore nella locazione di memoria (a 16 bit) puntata dal registro `W13`, riservato appunto per tale operazione di *Write-Back* delle istruzioni DSP;

5. incrementa di due l'indirizzo in W13;

Occorre notare che le istruzioni DSP possono essere configurate in modo da trattare esplicitamente i dati come numeri frazionari *Fixed-Point* con risoluzione fissata *Q15*. Ciò richiede di impostare a 1 il bit 0 dell'SFR **CORCON**. L'effetto di questa configurazione è che il risultato delle istruzioni di moltiplicazione viene automaticamente traslato verso sinistra di un bit. Come descritto nella Sezione B.2.4.2, ciò è necessario nell'aritmetica Fixed-Point perchè la moltiplicazione tra due numeri con un bit di segno e 15 bit di parte frazionaria fornisce un risultato con due bit di segno, uno dei quali è da scartare. Qualora il registro CORCON non sia impostato per l'uso della rappresentazione frazionaria, le istruzioni DSP sono trattate come normali operazioni tra numeri interi e non viene effettuato nessuno shift automatico del risultato.

Per completare la descrizione delle funzionalità tipiche di un DSP realizzate dai dsPIC33, si può evidenziare che il post-incremento degli indirizzi di X Data Memory e Y Data Memory, rispettivamente il terzo e quinto operando dell'istruzione MAC vista in precedenza, possono essere gestiti con la modalità *Modulo Addressing*. Per utilizzare tale modo di indirizzamento, necessario per l'accesso a buffer circolari, occorre:

1. abilitare la funzione di Modulo Addressing, in modo indipendente per X e Y Data Memory, tramite i bit 15 e 14 dell'SFR **MODCON**;

2. selezionare i puntatori desiderati per X e Y Data Memory, tramite i bit 0-3 e 4-7 del registro MODCON; per la prima area di memoria possono essere usati tutti i registri da W0 a W14, mentre per la seconda sono scelte valide solo W10 e W11; una ulteriore importante limitazione per l'uso della memoria Y è che il Modulo Addressing è ammesso solo nelle istruzioni DSP (i.e. MAC e simili) e solo con accesso read-only, mentre per la memoria X anche gli accessi da istruzioni della CPU base (es. `MOV`) sono gestiti con l'indirizzamento in modulo, se configurato;

3. definire l'intervallo di indirizzamento ammesso per i puntatori configurati, tramite gli SFR **XMODSRT, XMODEND, YMODSRT, YMODEND** rispettivamente inizio/fine del buffer in X Data Memory e inizio/fine del buffer in Y Data Memory.

Un'aspetto negativo del nucleo DSP dei processori di Microchip, comune peraltro alla maggior parte dei DSP/DSC, è il limitato supporto fornito dai compilatori in linguaggio C. Per sfruttare pienamente l'efficienza delle istruzioni DSP è necessario programmare direttamente nel linguaggio Assembly specifico, eventualmente inserendo linee di codice Assembly all'interno di sorgenti in linguaggio C. Per fare questo, il compilatore Microchip **MPLAB XC16** ammette la sintassi:

```
1 asm("MAC W4*W6, A, [W8]+=2, W4, [W10]+=2, W6, [W13]+=2");
```

NOTA BENE: il DSP Engine dei dsPIC33 è ovviamente un valido strumento per l'esecuzione di algoritmi di controllo PID, per i motivi ampiamente descritti nella Sezione B.2.4.1. Per un approfondimento sull'implementazione di PID nel caso specifico dei dsPIC33, comunque rappresentativo dei DSP/DSC in generale, si può consultare il tutorial alla pagina http://microchip.wikidot.com/pwr3201:pid-basics e successive, che descrive passaggio per passaggio l'esecuzione dell'algoritmo PID nella forma *incrementale*, come implementato dalla DSP Library (in assembler, con API per chiamare le funzioni dal linguaggio C) fornita da Microchip, in un totale di 13 cicli macchina (325 ns a 40 MIPS).

Si noti infine che il DSP Engine supporta la saturazione dei valori negli accumulatori, secondo una logica configurabile tramite l'SFR CORCON e ben descritta da questa pagina: http://microchip.wikidot.com/dsp0201:overflow-and-saturation. L'operazione di saturazione è di fondamentale importanza per realizzare l'anti-windup del controllore PID (v. Sezione B.2.4.1). Tuttavia, la saturazione applicata dal dsPIC33, come peraltro in molti altri DSP/DSC, è riferita ai valori massimo e minimo rappresentabile con l'aritmetica Fixed-Point supportata. Ad esempio, la saturazione applicata su valori in formato Q31 (o 1.31 nella terminologia usata dalla documentazione Microchip) limita l'uscita del PID tra -1.0 e 0.999969482422. Sarà poi compito del programmatore scalare l'uscita in modo opportuno per riportare tale intervallo a quello di interesse per gli scopi del controllo, cioè quello del valore di comando degli attuatori e/o dei convertitori di potenza (es. duty cycle di un segnale PWM).

B.2.5.2 Periferiche del dsPIC33FJ128MC802

Le periferiche integrate nei dsPIC33 ed in particolare nel dispositivo qui considerato sono particolarmente orientate alle applicazioni di controllo per azionamenti elettrici[31]. Oltre alle funzionalità più avanzate, è interessante osservare anzitutto il circuito di configurazione delle porte di I/O, mostrato in Figura B.2.38.

Tale circuito è simile a quello dei PIC a 8 bit (v. Figura B.2.19), ma con due importanti differenze.

1. Come nei PIC, le porte sono indicate con lettere dell'alfabeto e sono presenti un latch associato ai registri **TRISx** (a 16 bit, **x** indica la lettera della porta), per l'impostazione della direzione della porta (bit a 1 = Input, bit a 0 = Output), ed un latch di impostazione dello stato logico, che può essere però comandato tramite due registri: **LATx** o **PORTx**. La differenza tra questi due registri è che in lettura il secondo corrisponde direttamente al livello di tensione sul pin. È perciò consigliabile usare il registro TRISx per scrivere lo stato delle porte configurate come Output ed il registro PORTx per leggere lo stato delle porte configurate come Input.

2. L'impostazione come Output della porta può essere controllata direttamente dall'abilitazione di una determinata periferica (porte ***module controlled***). In questo caso, la configurazione della periferica connessa ad un certo pin prevale sul bit di registro TRISx corrispondente.

Considerando la sintassi ammessa dal compilatore MPLAB XC16, l'accesso agli SFR per le porte di I/O e in generale a tutti gli altri SFR, può essere fatto in tre modi:

[31] Per la descrizione completa delle periferiche dei dsPIC30/33, non è in genere sufficiente consultare il datasheet specifico di uno dei dispositivi, nel caso qui considerato scaricabile da https://www.microchip.com/wwwproducts/en/dsPIC33FJ128MC802, ma è necessario consultare anche i capitoli specifici, ricchi di commenti ed esempi, dei **Reference Manuals** validi per l'intera famiglia, scaricabili dalla pagina citata nella sezione *Documentation*.

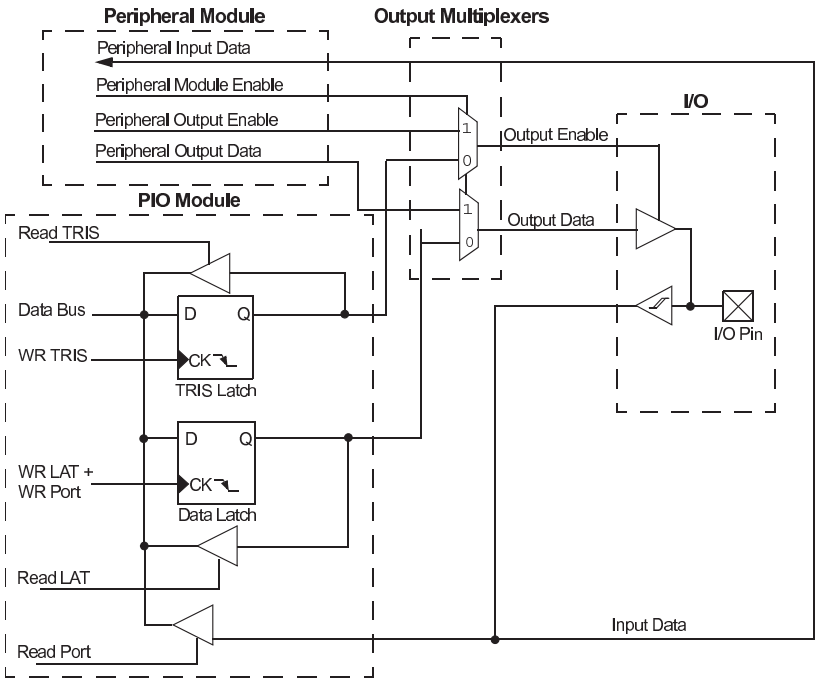


Figura B.2.38: Circuito tipico delle porte di I/O nei dsPIC33

1. tramite l'intera word a 16 bit:

```
TRISB = 0xFFDF;      // Pin RB5 = Output, all others pin RBx
LATB  = 0x0020;      // Set RB5 high
if(PORTB & 0x0040) // Test if RB6 is high
```

2. tramite struttura dati con bit (o gruppi di bit) predefiniti:

```
TRISBbits.TRISB5 = 0; // Pin RB5 = Output
LATBbits.LATB5   = 1; // Set RB5 high
if(PORTBbits.RB5)     // Test if RB6 is high
```

3. tramite nome specifico del bit (con prefisso _):

```
_TRISB5 = 0;   // Pin RB5 = Output
_LATB5  = 1;   // Set RB5 high
if(_RB6)       // Test if RB6 is high
```

Inoltre, molti dei pin del dsPIC33FJ128MC802 sono **rimappabili (Peripheral Pin Select, PPS)**, cioè permettono di definire via software l'associazione con una determinata periferica. Nei PIC a 8 bit, invece, il set di periferiche multiplexate su un certo pin è predefinito. Lo schema per la rimappatura delle porte di I/O nei dsPIC33 è mostrato in Figura B.2.39. I registri di configurazione per la rimappatura degli **Input** sono **associati alle periferiche**. Ad esempio, l'ingresso di ricezione della UART1 (**U1RX**) è definito tramite il contenuto del registro **RPINR18**, il cui valore indica il numero del pin rimappabile configurato (es. 11 per **RP11**). Inotre, è anche ammessa l'associazione di più periferiche allo stesso pin di input, funzionalità utile ad esempio per l'uso simultaneo di un modulo di conteggio (*QEI*) ed uno di *Input Capture* (per misure di velocità) con lo stesso segnale di un encoder incrementale. I registri di configurazione degli **Output** sono invece **associati ai pin**, come è ovvio dato che non sarebbe ragionevole associare le uscite di più periferiche allo stesso pin. Tutti i registri **RPORx** contengono due gruppi di cinque bit (**RPxR**), nei quali **x** indica il numero di un pin rimappabile ed il contenuto specifica quale periferica sia associata. Ad esempio, il registro RPOR0 include le configurazioni dei pin RP0/RP1 (bit RP0R e RP1R), RPOR1 i pin RP2/RP3 e così via.

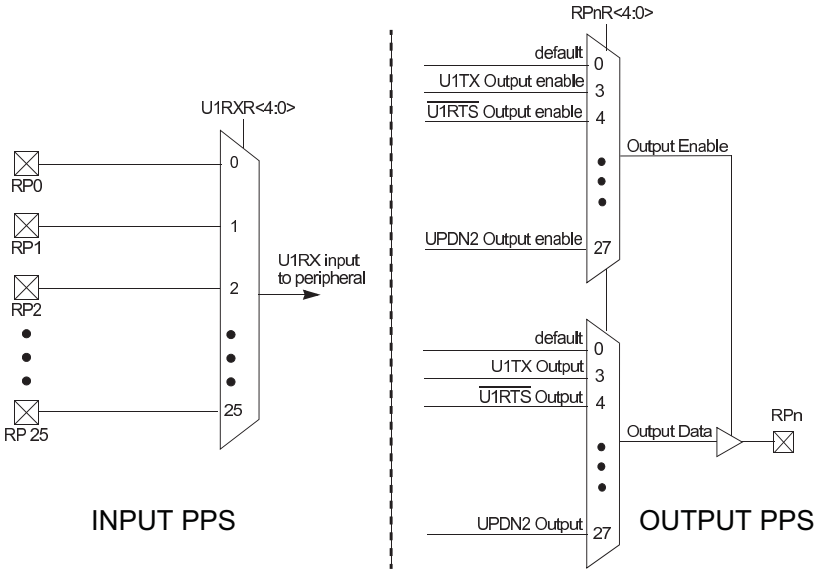


Figura B.2.39: Rimappatura delle periferiche (*Peripheral Pin Select, PPS*) sulle porte I/O nei dsPIC33

Timer/Counter

I moduli Timer/Counter nei dsPIC33 sono di tre tipi, denonimanti A, B e C, tutti a 16 bit. Quelli delle ultime due tipologie possono però essere concatenati per formare timer a 32 bit. Nel dsPIC33FJMC802 vi sono 5 timer: **Timer1** (tipo A), **Timer2/4** (tipo B) e **Timer3/5** (tipo C). La differenza principale degli ultimi due rispetto agli altri è la possibilità di generare il segnale trigger per la conversione A/D, come verrà descritto nella sezione successiva. Il principio di funzionamento dei timer di tipo C, valido anche in generale escludendo la parte di trigger ADC, è descritto in Figura B.2.40.

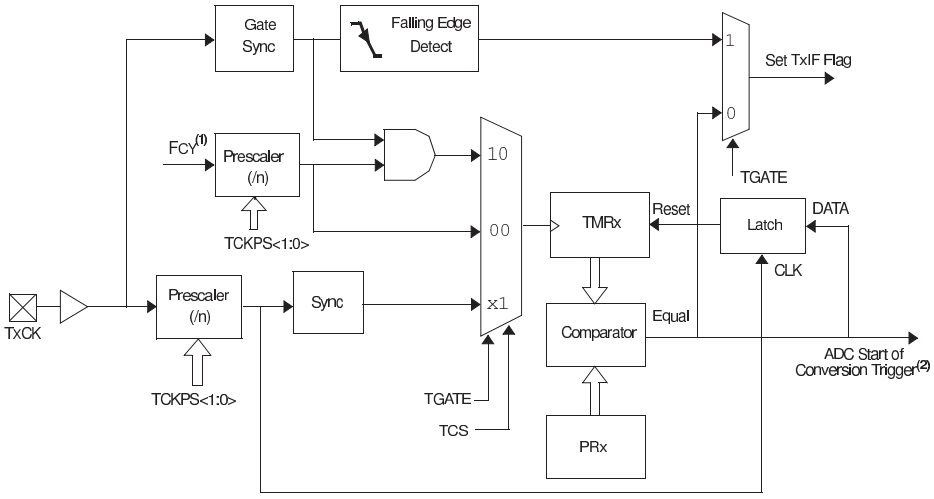


Figura B.2.40: Schema dei timer di tipo C nei dsPIC33

La configurazione dei timer si basa su un unico registro **TxCON** (**x** indica il numero del timer), con il quale si imposta se il segnale di clock sia prelevato da quello della CPU o da un pin esterno, si imposta un divisore per la frequenza di clock (**prescaler**), l'eventuale concatenazione a 32 bit

e si abilita il conteggio. Il valore di conteggio è memorizzato nel registro **TMRx** e viene resettato automaticamente (generando al contempo un interrupt, se abilitato) quando coincide con il valore del registro **PRx** (*period register*), che definisce di fatto il periodo di ripetizione del ciclo di conteggio.

Convertitore A/D

Il modulo ADC dei dsPIC33 è significativamente più complesso di quello normalmente integrato nei PIC a bit. Anzitutto, come mostrato in Figura B.2.41, è dotato di quattro diversi circuiti di Sample/Hold (S/H) che permettono di campionare simultaneamente fino a quattro segnali. Sebbene il convertitore A/D sia comunque unico, tale funzionalità è utile per acquisire misure che richiedono il campionamento sincronizzato in precisi istanti di tempo (es. correnti di fase in un motore elettrico).

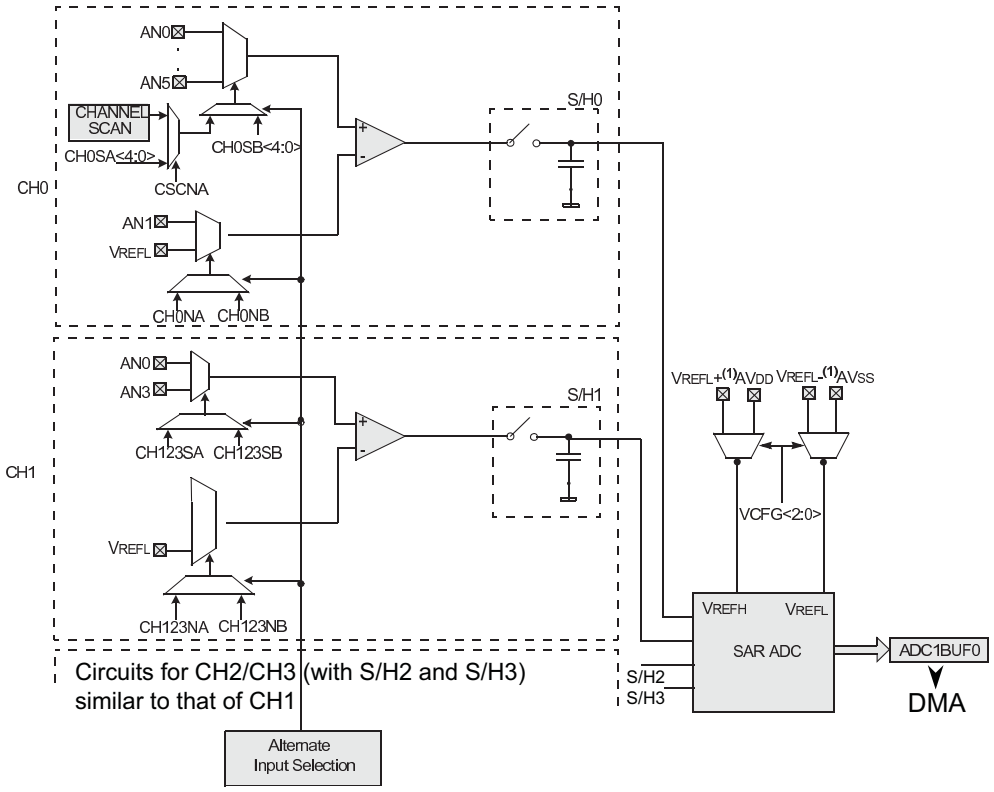


Figura B.2.41: Schema del convertitore A/D del dsPIC33FJ128MC802

La grande versatilità dell'ADC è dimostrata anche dalla presenza di ben 8 SFR a 16 bit per la configurazione:

- **AD1CON1**, per abilitare il modulo, impostare la conversione a 10 o 12 bit e il formato del risultato come intero o fractional, abilitare il campionamento simultaneo dei canali S/H e selezionare il segnale di trigger per la conversione. Questo può essere, oltre al flag **SAMP** dello stesso registro:
  1. un fronte di salita sul pin di interrupt esterno **INT0**;
  2. il **Timer3** o il **Timer5**, al raggiungimento di un determinato valore di conteggio;
  3. il modulo **Motor Control PWM**, ad un fissato istante del periodo PWM.
- **AD1CON2**, permette di selezionare le tensioni di riferimento per l'ADC, se campionare simultaneamente tutti i quattro canali S/H (**CHS0/1/2/3**), solo i primi due oppure solo **CH0** e, in quest'ultimo caso, se effettuare la scansione ciclica automatica dei segnali multiplexati

su tale canale S/H. Inoltre, permette di definire dopo quante conversioni effettuate in automatico venga generato un interrupt per la CPU, da una fino ad un massimo di 16. Per il dsPIC33FJ128MC802, se viene definito un numero maggiore di uno per tale parametro è necessario configurare anche il trasferimento dei risultati di conversione dal buffer temporaneo **AD1BUF0** (a 16 bit) alla RAM tramite DMA.

- **AD1CON3**, per impostare il clock del convertitore ad approssimazioni successive (la cui operazione richiede tanti cicli di clock quanti sono i bit del risultato più due).
- **AD1CON4**, per definire la dimensione del buffer di scambio dati con il DMA.
- **AD1CHS0 e AD1CHS123**, il secondo imposta i pin collegati ai S/H CHS1/2/3, utilizzati effettivamente solo se è abilitato il campionamento simultaneo, mentre il primo configura il multiplexer per CH0.
- **AD1CSSL**, per selezionare i pin per la scansione ciclica dei segnali su CH0, se abilitata.
- **AD1PCFGL**, per configurare i pin associati ai bit di registro in Analog mode o Digital mode.

La modalità di campionamento più semplice, cioè l'attivazione "manuale" della conversione e l'attesa del termine dell'operazione con *polling* del flag di stato, si traduce in linguaggio C come segue:

```
// Initialize MUXA Input Selection
AD1CHS0bits.CH0SA = 3;      // Select AN3 for CH0 +ve input
AD1CHS0bits.CH0NA = 0;      // Select VREF- for CH0 -ve input
// Sample and conversion
AD1CON1bits.SAMP = 1;       // Start sampling
__delay_us(10);             // Wait for sampling time (10us)
AD1CON1bits.SAMP = 0;       // Start the conversion
while (!AD1CON1bits.DONE);// Wait for the conversion to complete
ADCValue = ADC1BUF0;        // Read the conversion result
```

Nelle configurazioni più complesse, ad esempio quella con quattro campionamenti simultanei e su ingressi analogici alternati fra due diverse configurazioni dei multiplexer, è necessario il DMA per la bufferizzazione dei risultati. La Figura B.2.42 mostra la sequenza operativa dell'ADC in combinazione con il DMA, con impostazione ottenuta dal codice C:

```
AD1CON1bits.AD12B = 0;      // Select 10-bit mode
AD1CON2bits.CHPS = 3;       // Select 4-channel mode
AD1CON1bits.SIMSAM = 1;     // Enable Simultaneous Sampling
AD1CON2bits.ALTS = 1;       // Enable Alternate Input Selection
AD1CON2bits.SMPI = 1;       // Two sample/conversion cycles per DMA
                            // address increment
AD1CON1bits.ASAM = 1;       // Enable Automatic Sampling
AD1CON1bits.SSRC = 2;       // Timer3 generates SOC trigger
// Initialize MUXA Input Selection
AD1CHS0bits.CH0SA = 6;      // Select AN6 for CH0 +ve input
AD1CHS0bits.CH0NA = 0;      // Select VREF- for CH0 -ve input
AD1CHS123bits.CH123SA = 0; // Select CH1 +ve = AN0, CH2 +ve = AN1, CH3 +ve = AN2
AD1CHS123bits.CH123NA = 0; // Select VREF- for CH1/CH2/CH3 -ve inputs
// Initialize MUXB Input Selection
AD1CHS0bits.CH0SB = 7;      // Select AN7 for CH0 +ve input
AD1CHS0bits.CH0NB = 0;      // Select VREF- for CH0 -ve input
AD1CHS123bits.CH123SB = 1; // Select CH1 +ve = AN3, CH2 +ve = AN4, CH3 +ve = AN5
```

Modulo Input Capture

Nei dsPIC33, le funzioni tipicamente usate per il controllo di motori elettrici, cioè *Input Capture* per misure di durata di impulsi digitali (e rilevare la velocità di encoder incrementali) e *Output Compare* per la generazione di impulsi PWM, sono realizzata da periferiche indipendenti. Lo schema di funzionamento della prima tipologia è descritto dalla Figura B.2.43.

La configurazione di tale modulo nel dsPIC33FJ28MC802 richiede:

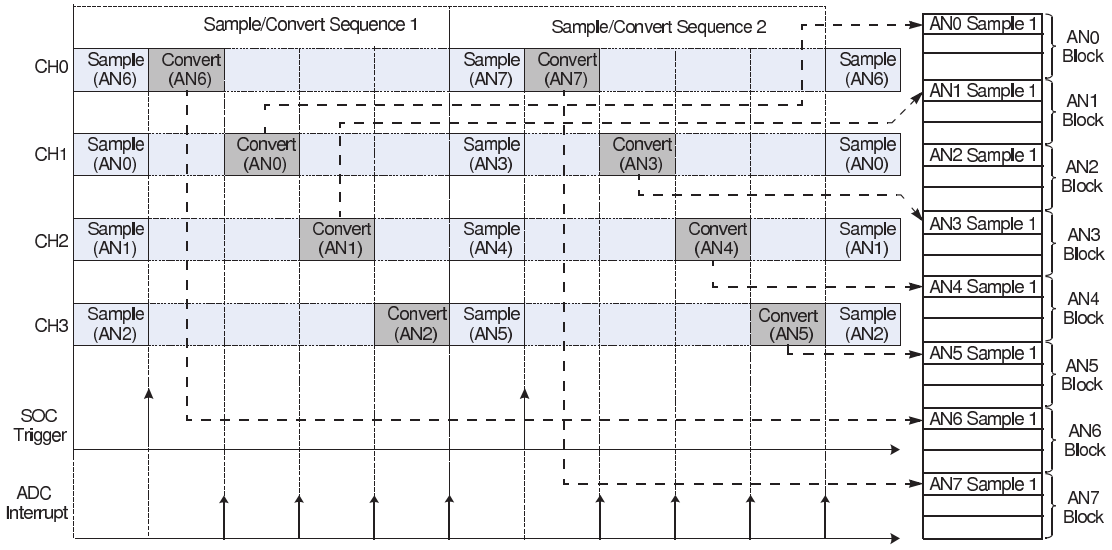


Figura B.2.42: Sequenza di campionamento simultaneo e conversione alternata, con trasferimenti dati via DMA

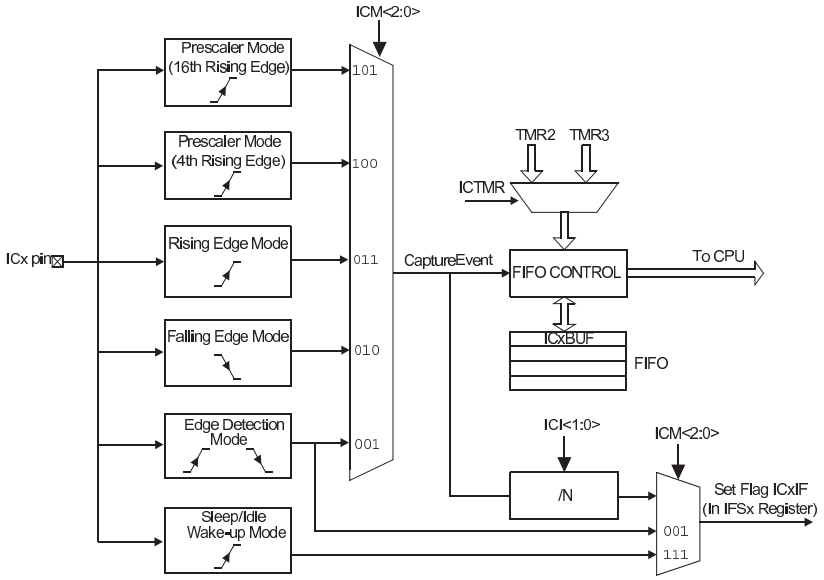


Figura B.2.43: Schema del modulo Input Capture del dsPIC33FJ128MC802

1. l'impostazione dell'SFR **ICxCON** (**x** indica il numero del modulo usato, 1 o 2), che permette di selezionare di Timer2 o Timer3 come base dei tempi, di definire dopo quanti fronti di salita o discesa catturare il valore del timer e dopo quanti eventi di *Capture* generare un interrupt per la CPU;
2. la configurazione e attivazione opportuna del timer selezionato, Timer2 o Timer3;
3. l'associazione della periferica ad un pin rimappabile, tramite la funzionalità di Peripheral Pin Select (PPS).

Modulo Quadrature Encoder Interface

Il modulo **Quadrature Encoder Interface** (QEI) effettua il conteggio di segnali digitali in quadratura, tipicamente generati dai sensori di tipo Encoder Incrementale (v. Sezione A.2.1.4 della Parte A di queste dispense). Il modulo effettua un filtraggio digitale dei segnali tipicamente indicati come A, B e INDEX (tacca di zero) negli Encoder, li decodifica per incrementare il successivo conteggio per 2 o per 4, incrementa/decrementa conseguentemente un contatore a 16 bit ed infine azzera il valore memorizzato dal contatore in corrispondenza del cambiamento di stato del segnale INDEX oppure quando il valore di conteggio stesso supera un determinato valore massimo (operazione utile per mantenere il conteggio all'interno di un giro meccanico, per Encoder rotativi NON dotati della tacca di zero). La Figura B.2.44 mostra lo schema a blocchi di tale modulo.

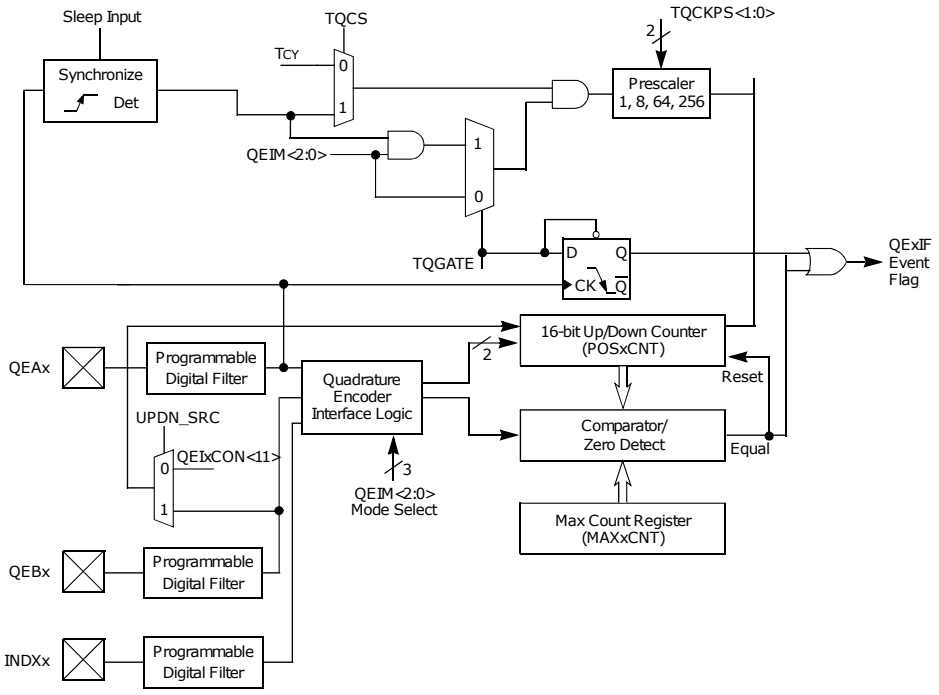


Figura B.2.44: Schema del modulo Quadrature Encoder Interface del dsPIC33FJ128MC802

La gestione del modulo QEI è svolta tramite i seguenti SFR (con **x**=1 o 2, se sono presenti due moduli QEI):

- **QEIxCON**: per la configurazione della modalità operativa;
- **DFLTxCON**: per configurare i filtri sui pin di input;
- **POSxCNT**: valore di conteggio corrente;
- **MAXxCNT**: valore massimo in corrispondenza del quale azzerare POSxCNT.

Il filtraggio digitale eseguito dal modulo QEI del dsPIC33 corrisponde più precisamente ad una serie di circuiti flip-flop all'uscita dei quali il valore logico dei canali A, B e INDEX cambia stato solo se i corrispondenti pin di ingresso rimangono ad un livello stabile per più cicli di clock, allo scopo di eliminare impulsi spuri eventualmente causati da interferenze elettromagnetiche (tipiche per gli Encoder installati su motori elettrici).

Oltre che come contatore di segnali da Encoder, il modulo QEI può anche essere utilizzato come Timer/Counter a 16 bit standard, nel qual caso il valore di POSxCNT viene incrementato in corrispondenza del clock di sistema oppure dei fronti sul pin A associato al modulo e il valore di MAXxCNT viene utilizzato come *period register*. Infine, il modulo può generare una causa di interrupt se:

- è rilevato un fronte sul segnale INDEX (e contestualmente è abilitato l'azzeramento del conteggo su tale condizione);
- è configurato l'azzeramento del contatore POSxCNT sulla condizione di matching con MAXxCNT e tale condizione si verifica;
- il modulo è impostato come Timer/Counter standard e POSxCNT raggiunge il valore di MAXxCNT.

Modulo Motor Control PWM

La generazione di impulsi PWM nei dsPIC33, oltre che con il modulo *Output Compare* (qui non considerato), può essere gestita in modo molto efficiente grazie al modulo **Motor Control PWM** (MCPWM), specifico per il controllo di motori elettrici, il cui schema funzionale è mostrato in Figura B.2.45.

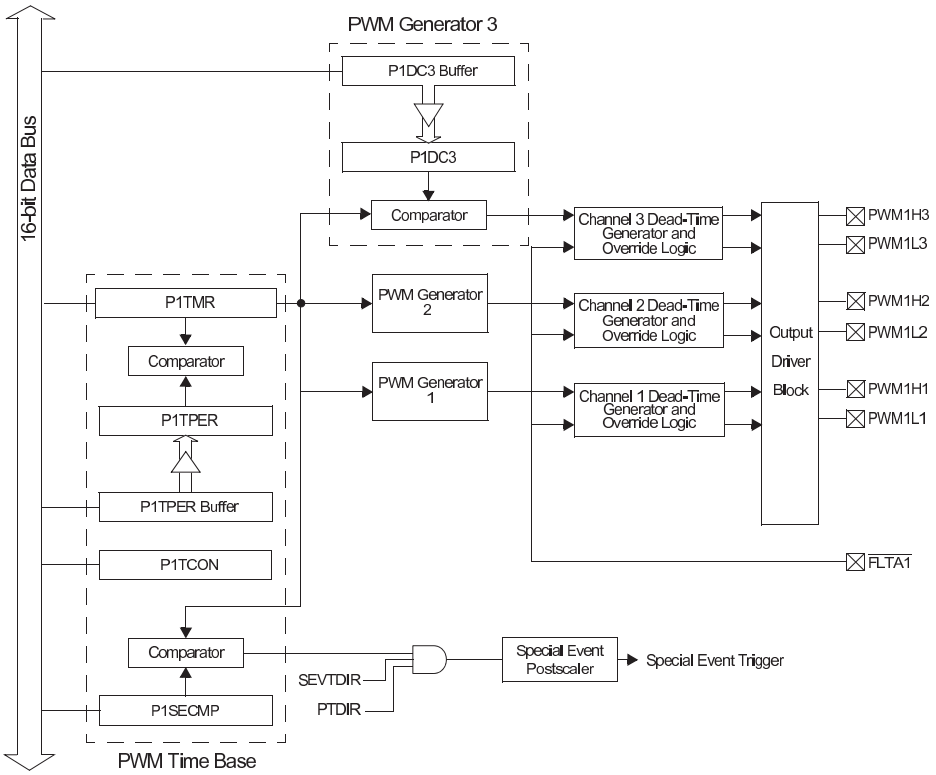


Figura B.2.45: Schema del modulo Motor Control PWM del dsPIC33FJ128MC802

Nel dsPIC33FJ128MC802 sono integrati due di questi moduli, uno con 3 generatori per pilotare 6 pin (**PWM1H/L1, PWM1H/L2, PWM1H/L3**) ed uno con un solo generatore per pilotare due pin (**PWM2H/L1**). Tali pin sono accoppiati a due a due per effettuare la modulazione complementare dei transistor **High-Side** e **Low-Side** nelle tipiche configurazioni a ponte dei convertitori di potenza *switching* (si veda ad esempio Figura B.2.35 con la topologia di un inverter trifase).

In generale, la modulazione di impulsi PWM richiede un timer ed un comparatore che confronti il valore di conteggio con il valore della grandezza numerica richiesta per il Duty Cycle. Il modulo MCPWM non necessita di selezionare uno dei timer base del dsPIC33, ma ne utilizza uno dedicato con possibilità di conteggio *up-down*. Infatti, qualora si debbano sincronizzare più segnali PWM con lo stesso periodo, i modulatori più evoluti come quello dei dsPIC33 possono essere configurati per ottenere impulsi **asimmetrici** (la durata di attivazione del pin T_{on} è sempre "allineata" all'inizio

del periodo T) o **simmetrici** (il tempo T_{on} è allineato al centro di T), come schematizzato in Figura B.2.46. La seconda configurazione riduce le emissioni di disturbi elettromagnetici nei convertitori di potenza in quanto, a differenza della prima, evita l'accensione simultanea di tutti i transistor comandati dai segnali PWM.

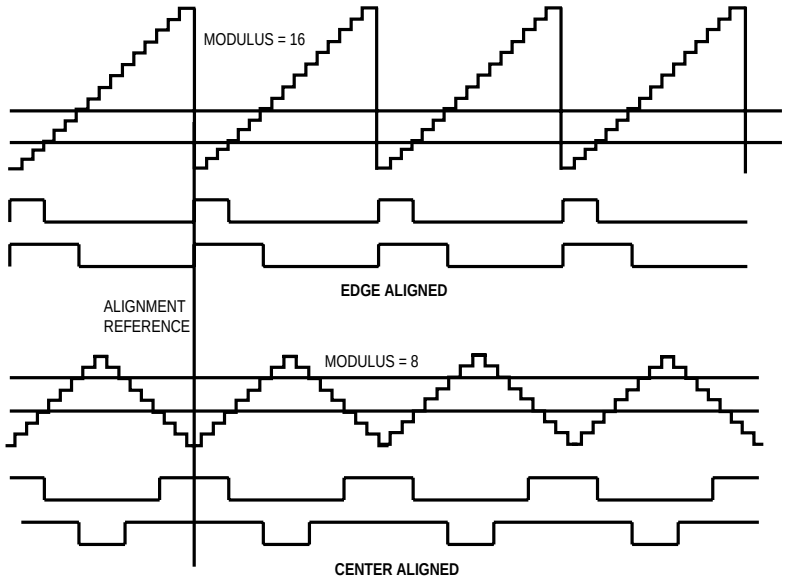


Figura B.2.46: Forme d'onda asimmetriche (sopra) e simmetriche (sotto) per impulsi PWM

Un'altra funzionalità molto importante nel controllo dei convertitori di potenza a ponte, con transistor complementari *High/Low-Side*, è l'inibizione automatica degli stati di corto-circuito, nei quali cioè risultino contemporaneamente accesi i transistor posti su uno stesso ramo del ponte. Per evitare tali configurazioni, il modulo McPWM può essere configurato per ritardare con un certo **Dead-Time** l'accensione di un pin PWMxHy rispetto allo spegnimento del corrispondente pin PWMxLy e viceversa, come mostrato nella Figura B.2.47 per una generica applicazione.

Infine, come già citato nella descrizione del convertitore A/D, il modulo MCPWM può generare il segnale di trigger per il campionamento. Il grande beneficio di tale funzionalità si manifesta nelle applicazioni che richiedono la regolazione in retroazione della corrente in un carico induttivo (es. motori elettrici, valvole elettromagnetiche, ecc.). Infatti, in tali applicazioni la corrente è soggetta ad ondulazioni implicite, dovute alla stessa modulazione PWM. Configurando il generatore di PWM con impulsi simmetrici e campionando il feedback di corrente al centro del periodo, tale istante corrisponde anche al centro della rampa di salita della corrente, la cui misura è pertanto meno influenzata dalle oscillazioni di modulazione. Come mostrato in Figura B.2.48, con il campionamento NON sincronizzato rispetto al periodo PWM, l'acquisizione della corrente risulterebbe molto meno stabile e richiederebbe una successiva elaborazione con filtraggio passa-basso a frequenza inferiore a quella del segnale di comando PWM, filtraggio non necessario invece con al campionamento sincronizzato.

Essendo il modulo MCPWM particolarmente complesso, la sua configurazione richiede numerosi SFR:

- **PWMxCON1**, seleziona le modalità indipendenti o complementari per ogni coppia di pin PWM, oppure ne definisce l'uso come I/O standard;
- **PWMxCON2**, imposta il postscaler per il *PWM Special Event Trigger* (per il campionamento A/D) ed abilita alcune modalità di aggiornamento dei valori di Duty Cycle;
- **PxDTCON1/2**, impostano il clock ed i valori di *Dead Time*;

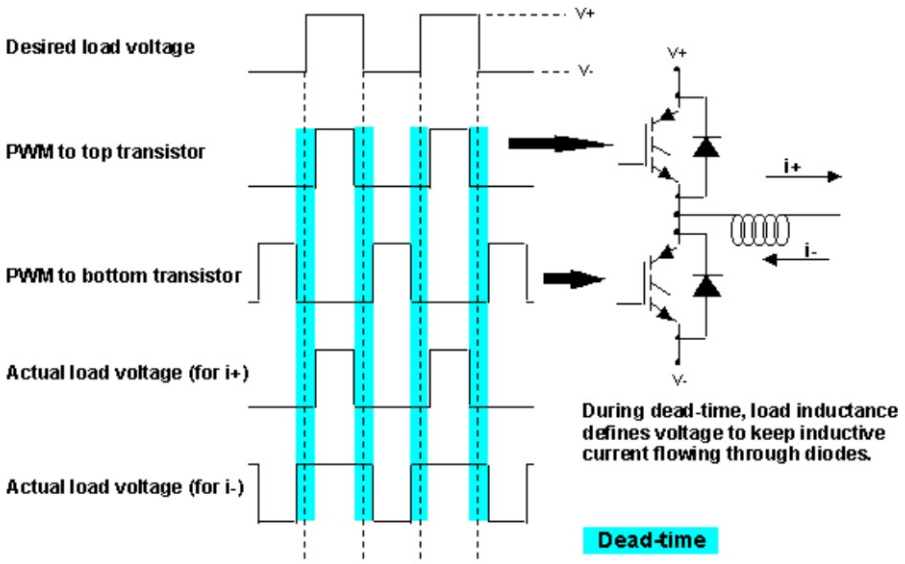


Figura B.2.47: Inserimento di *dead-time* nel comando PWM di transistor complementari in un convertitore di potenza (fonte http://www.mcjournal.com/articles/arc107/arc107.htm)

- **PxFLTA/BCON**, impostano le modalità di disattivazione della generazione PWM in risposta al cambiamento di stato dei pin di **Fault** specifici, tipicamente connessi ad appositi circuiti di monitoraggio con soglia delle correnti nei convertitori di potenza;
- **PxOVDCON**, abilita la funzionalità di **Override**, che permette di forzare lo stato dei pin PWM in modo indipendente dal modulatore;
- **PxTCON**, configura il timer dedicato **PWM Time Base**, il cui valore di conteggio è memorizzato nel registro a 16 bit **PxTMR** e resettato quando corrisponde al valore di fine periodo, impostato nel registro **PxTPER**. Inoltre, in corrispenza del valore impostato nel registro **PxSECMP**, il timer genera il trigger per l'ADC, se configurato.

I valori corrispondenti al Duty Cycle sono memorizzati nei registri **PxDC1/2/3**, uno per ogni coppia di pin PWM *High/Low-Side*. Poichè il PWM Time Base si resetta al valore impostato da PxTPER e il generatore PWM effettua il confronto con il timer due volte per ciclo di clock per determinare l'istante di cambiamento di stato dei pin PWM, il Duty Cycle al 100% corrisponde al valore di **PxTPER** $\times$ **2**.

Infine, è importante osservare che oltre agli SFR citati, esiste un registro non modificabile via software chiamato **FPOR**, il cui valore viene assegnato solo all'atto della programmazione della memoria Flash del dsPIC con il codice eseguibile. Tale tipo di registri viene definito **Device Configuration Registers** o **Configuration Bits**. Nel caso specifico di FPOR, esso serve per impostare la polarità dei pin PWM, cioè se i segnali vengano generati con logica positiva (livello alto durante il periodo T_{on}) o negativa (livello basso durante il periodo T_{on}).

B.2.5.3 Gestione interrupt nel dsPIC33FJ128MC802

La gestione degli interrupt nei dsPIC33 è molto articolata. Le caratteristiche dell'Interrupt Controller del dsPIC33 si possono riassumere come segue:

- Definisce 16 livelli di priorità, otto dei quali riservati per eventi non mascherabili definiti **Traps**, associati ad errori critici della CPU. Il controllo dell'esecuzione viene passato solo ad eventi di interrupt con priorità maggiore di quella del contesto corrente.

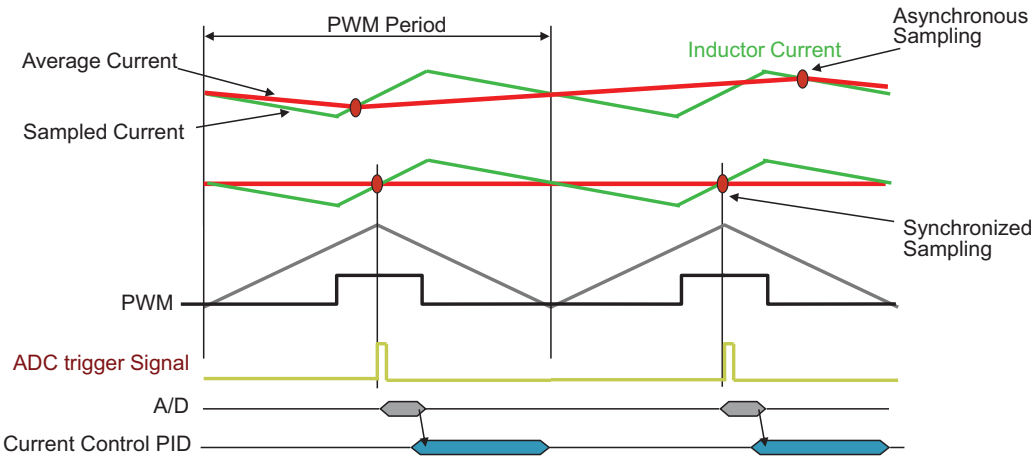


Figura B.2.48: Benefici della conversione A/D sincronizzata con i segnali di controllo PWM

- Oltre che quella delle ISR, anche la priorità della CPU stessa (i.e. quella del programma principale) è sempre configurabile via software dal livello 0 (inferiore) al 7. Ciò significa che è possibile disabilitare in certe parti del codice gli interrupt provenienti da eventi con priorità inferiore, mantenendo attive le sorgenti di interrupt più prioritarie.
- Definisce due diverse **Interrupt Vector Table** (IVT e *Alternate* IVT, AIVT), ciascuna con 126 indirizzi di memoria codice per le ISR. La selezione di quale delle due tabelle IVT usare può essere cambiata via software, il che permette di definire due modalità diverse di gestione dello stesso evento di interrupt in base allo stato del programma.
- L'ordine degli eventi di interrupt nella IVT definisce anche la priorità implicita tra gli stessi, utilizzata per risolvere conflitti di esecuzione nel caso in cui venga assegnata la stessa piorità a più eventi abilitati.

Gli SFR per la configurazione degli interrupt sono:

- **INTCON1**, contiene il bit *Interrupt Nesting Disable* (**NTDIS**) e i flag di stato degli eventi di *Trap*;
- **INTCON2**, imposta i fronti di interesse per i pin di interrupt esterno e seleziona IVT o AIVT;
- **IEC0-4**, contengono tutti i bit di abilitazione dei vari eventi di interrupt;
- **IPC0-19**, contengono le impostazioni di priorità degli eventi di interrupt;
- **IFS0-4**, contengono i flag di stato degli eventi di interrupt;
- **SR (Status Register)**, oltre ad altri bit non usati dall'Interrupt Controller, imposta il livello di priorità della CPU.

Infine, la sintassi ammessa dal compilatore MPLAB XC16 per la definizione delle ISR è la seguente:

```
1 void __attribute__((interrupt)) _T1Interrupt(void)
2 {
3   IFS0bits.T1IF = 0; // il flag di stato va resettato manualmente
4
5   ...
6 }
```

Si noti anche i nomi delle ISR devono combaciare con quelli definiti nel file *Linker Script* corrispondente al dispositivo target (i.e. `p33FJ128MC802.gld` per il DSC qui considerato), nella sotto-cartella `../support/dsPIC33F/gld` del percorso di installazione del compilatore MPLAB XC16.

Capitolo B.3

Sistemi Configurabili e Distribuiti

La progettazione di un sistema automatizzato complesso è un compito solitamente realizzato in base ad una ben precisa richiesta da parte di un singolo committente, il quale molto spesso necessita di un esemplare unico o comunque molto specializzato. Pertanto, il sistema di controllo per tale tipo di applicazioni deve essere progettato in modo:

- configurabile
- espandibile
- modulare (in modo da poter riparare o sostituire agevolmente ogni singola sottoparte)
- economico (costi e tempo di progettazione minimi).

Per tutti questi motivi, la scelta ottimale nella realizzazione di un sistema di controllo per uno specifico impianto automatizzato prevede architetture basate sui seguenti elementi:

- Diversi sistemi di elaborazione, collegati tra loro almeno con linee elettriche per lo scambio di semplici segnali digitali di sincronizzazione oppure, per ottenere funzionalità più evolute, con specifiche linee di comunicazione elettriche;
- Ciascun sistema di elaborazione è a sua volta costituito da un certo numero di schede elettroniche separate, prontamente disponibili sul mercato (*"off-the-shelf"*), ciascuna dedicata ad un determinato sotto-compito (elaborazione, conversione A/D e D/A, comunicazione con pannelli operatore, ecc.), collegate da un **bus parallelo**.

Ovviamente, tale tipo di architettura coincide con quella dei sistemi informativi tipicamente usate per l'elaborazione dati "office", sebbene in tal caso vengano realizzati con componenti e funzionalità totalmente differenti. Nel seguito verranno descritte brevemente le caratteristiche che distinguono le architetture modulari e distribuite di sistemi per il controllo.

B.3.1 Sistemi configurabili: Bus standard e proprietari

Un generico bus parallelo che permette la connessione di schede elettroniche in modo espandibile e riconfigurabile viene detto **Backplane Bus**. Un *backplane bus* è caratterizzato dal numero di bit di dato (16, 32 o 64 bit per quelli più evoluti,), dalla massima velocità di trasmissione in MBytes/s e ovviamente dal protocollo di scambio dati e arbitraggio dell'accesso alla linea elettrica.

I sistemi con architetture a bus parallelo, si distinguono poi in base al fatto che il bus stesso sia **standardizzato** o di tipo **proprietario**. Nel primo caso, le specifiche elettriche e sul protocollo di interfacciamento e comunicazione sono regolate da documenti internazionali accessibili a qualunque produttore, in modo tale che vi possa essere un'ampia scelta sul mercato di dispositivi compatibili con quell'architettura. Nel secondo caso, invece, le caratteristiche del bus sono note solamente al costruttore che le ha progettate o al più a terze parti ufficialmente accreditate.

I **bus standard** utilizzati nei sistemi industriali per il controllo coincidono funzionalmente con quelli utilizzati per le schede di espansione dei PC per uso "office", cioè **PCI/PCIexpress**. Esistono tuttavia delle varianti che rendono le schede elettroniche compatibili più idonee alle applicazioni industriali e di controllo, soprattutto grazie al formato dei connettori e delle schede stesse:

- **PC/104 e PCI/104**: il primo nasce come adattamento del bus **ISA**, attualmente in disuso nei Personal Computer, mentre il secondo è funzionalmente compatibile con il protocollo **PCI**. Il formato del connettore e delle schede stesse è però molto più compatto (9 × 9.6 cm, le dimensioni di un floppy-disk) e soprattutto queste sono connesse tra di loro direttamente l'una all'altra (a "stack", perciò senza un "backplane"), cosicchè si può ottenere un PC completo con processore, memorie e schede di interfaccia in dimensioni ridottissime, come evidenziato nella Figura B.3.1.

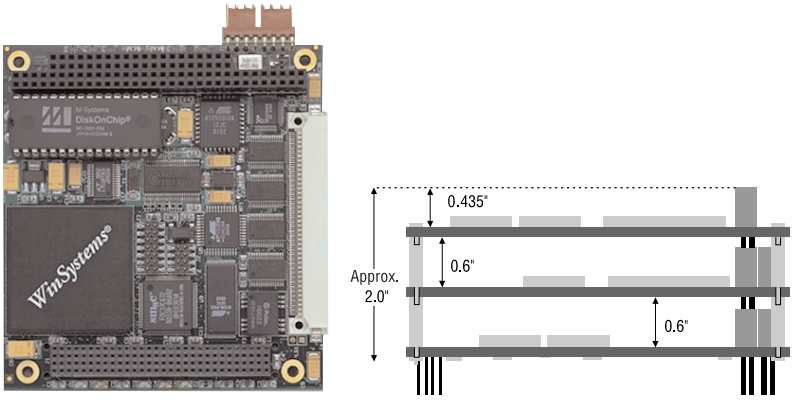


Figura B.3.1: Scheda per bus PC-PCI/104 e connessione in "stack" fra tali schede

- **VME:** è un bus storicamente molto usato nei sistemi di acquisizione dati e di controllo, in quanto all'epoca della sua introduzione (1981), aveva caratteristiche molto superiori ai concorrenti derivati da architetture PC. Inizialmente lo standard prevedeva 4 diversi formati delle schede, tra 16 × 10 cm e 34 × 36 cm, parallelismo a 8, 16 e 32 bit e velocità di comunicazione fino 40 MB/s. La versione più moderna, chiamata **VME64** (a 64 bit), può arrivare a 80 MB/s ed è ancora discretamente utilizzata soprattutto nelle applicazioni **safety-critical**.

Figura B.3.2: Un cestello (*rack*) per schede VME ed una scheda in formato 34 × 23 cm.

Una tipica applicazione del bus VME è storicamente quella del controllo di robot manipolatori, secondo schemi architetturali come quello mostrato in Figura B.3.3. In tale contesto infatti, oltre alla disponibilità di hardware specifico per l'acquisizione di segnali di misura (eg. encoder, correnti elettrice ecc.), il bus VME offre la possibilità di far comunicare tra loro diverse schede

CPU, ciascuna dedicata all'elaborazione ad alta frequenza di specifici algoritmi per il controllo del robot (es supervisione e interpretazione del programma di movimento definito dell'utente, calcolo di cinematica e dinamica inversa, generazione di traiettorie e controllo ad inseguimento, ecc.).

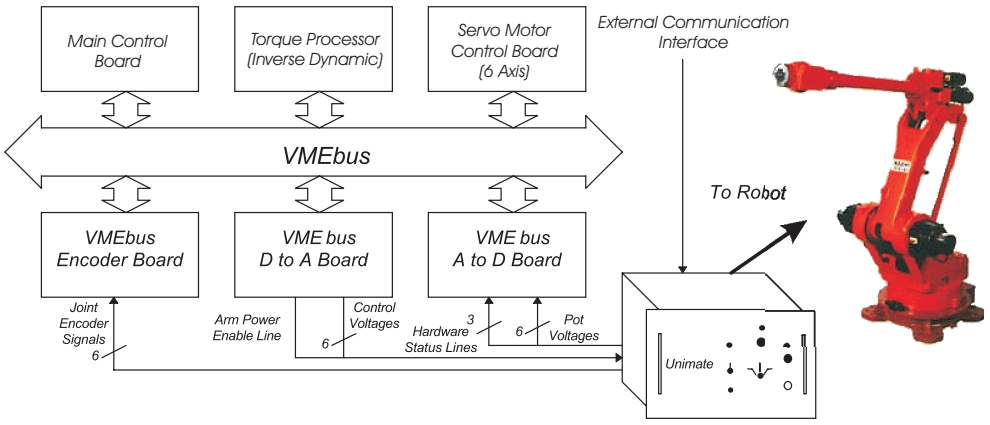


Figura B.3.3: Controllore per robot su bus VME

- **CompactPCI:** è caratterizzato dallo standard elettrico e di comunicazione del bus PCI, trasportato sullo standard fisico (dimensioni e pinout) del bus VME, nato per rimpiazzare quest'ultimo nelle applicazioni di controllo e acquisizione dati.

B.3.1.1 Applicazioni industriali basate su PC

Il Personal Computer, opportunamente predisposto con schede specifiche per l'acquisizione dati e per il controllo, è certamente appetibile per applicazioni industriali, grazie alla grande quantità di software ed hardware disponibile in commercio a prezzi bassi, la potenza di calcolo, le capacità di elaborazione numerica Floating-Point e la possibilità di programmazione con linguaggi evoluti (C/C++, Java, ecc.).

Di contro questa scelta ha un certo numero di limitazioni dovute al fatto che i PC per uso "office", sia dal punto di vista dei Sistemi Operativi, sia dal punto di vista hardware, non sono progettati per applicazioni di controllo. In particolare, il sistema operativo è in genere programmato per gestire "equamente" tutti i processi software, con il risultato che quelli che devono accedere a dispositivi "lenti" (disco rigido, scheda di rete, ecc.) possono occupare il sistema per tempi relativamente lunghi, rendendo impossibile l'acquisizione dati dal campo rispettando i di tempi di campionamento in modo rigido. Altre limitazioni sorgono da vincoli costruttivi: il numero di locazioni disponibili (*slot*) per schede di espansione è limitato, ed un PC prodotto per l'ufficio non è idoneo a lavorare nelle condizioni ambientali avverse (presenza di polvere, umidità, etc.) tipiche dell'ambiente industriale

I PC progettati per uso industriale, compatibili in termini di hardware e software con i PC, superano i vincoli sopra citati, ma con un notevole incremento dei costi. I PC per l'industria sono dotati di robusti contenitori, molti slot e connettori di espansione (v. Figura B.3.4) e sono programmati con sistemi operativi orientati all'elaborazione **real-time** (**RTOS**, Real-Time Operating Systems, più complessi e sofisticati di quelli già citati per il contesto dei microcontrollori in quanto in grado di garantire vincoli temporali anche nella gestione di operazioni su file, audio/video e perifiche di comunicazione USB, Ethernet che tipicamente i sistemi a microcontrollore non supportano)). Tra i più noti esempi di RTOS per sistemi PC-based si possono citare **VxWorks** (utilizzato anche da NASA per il progetto Curiosity, il rover in esplorazione sul pianeta Marte)[1], **QNX**[2], **Xenomai**

[1] https://www.windriver.com/products/vxworks
[2] https://blackberry.qnx.com/en

(estensione dual-kernel per Linux)[3] o anche Linux standard, ma con kernel nel quale sia abilitata l'opzione **PREEMPT_RT**[4].

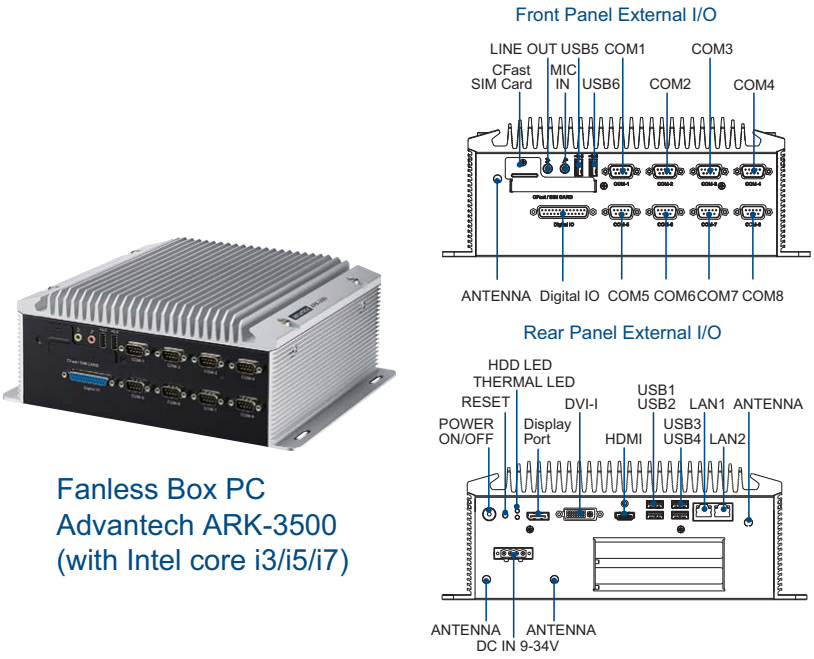


Figura B.3.4: Esempio di PC idoneo all'uso industriale

B.3.1.2 Sistemi a backplane bus *proprietario*: PLC

I dispositivi di controllo configurabili progettati in modo specifico per l'automazione industriale sono quasi sempre basati sulla integrazione delle schede elettroniche tramite backplane di tipo proprietario. I tipici dispositivi con queste caratteristiche sono i Controllori Logici Programmabili (**Programmable Logic Controllers, PLC**), componenti fondamentali nell'automazione per via della loro affidabilità, robustezza, capacità di espansione e semplicità di programmazione, eseguita principalmente mediante linguaggi di tipo grafico (Ladder Diagram, Sequential Function Chart, ecc.).

L'architettura di un PLC non si differenzia molto da quella di un calcolatore "general purpose" ed è molto simile ai sistemi di controllo analizzati in precedenza, infatti essa comprende (Figura B.3.5):

- il cestello o rack, che contiene e racchiude tutti gli altri moduli che compongono il PLC, assicurandone la connessione meccanica ed il collegamento elettrico.
- il modulo di alimentazione, che fornisce alimentazione ai moduli elettronici installati nel cestello.
- il **modulo processore (CPU)** (normalmente uno per rack), che esegue il programma di controllo e le altre elaborazioni necessarie al funzionamento del sistema.
- i **moduli di Input/Output digitali**, per l'acquisizione di informazioni da sensori logici ed il comando di interruttori e altri dispositivi a comando logico. Normalmente segnali a 0-24V.
- **I moduli di Input e di Output Analogici** per la conversione A/D (e D/A) di segnali di provenienza eterogenea, purchè codificati secondo il segnale elettrico predisposto (±10V,

[3] https://source.denx.de/Xenomai/xenomai/-/wikis/Start_Here
[4] https://rt.wiki.kernel.org/index.php/CONFIG_PREEMPT_RT_Patch

$4 \div 20$mA ecc.). Per semplificare le problematiche di acquisizione, esistono moduli speciali per certi tipi di sensore (es: termocoppie, con compensazione "giunzione fredda").

- **I moduli Funzionali Speciali** svolgono compiti di controllo dedicato, liberando la CPU principale da sovraccarichi di elaborazione o realizzando elaborazioni che essa non è in grado di eseguire. Ad esempio: **moduli di Conteggio Encoder**, **moduli di Motion Control**, **moduli di comunicazione per I/O distribuiti.**

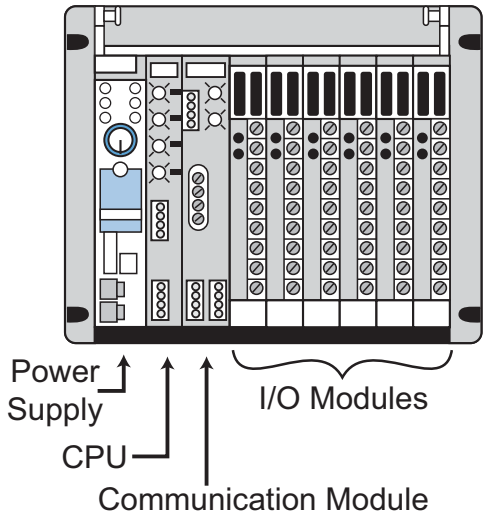


Figura B.3.5: Architettura di base di un PLC.

Anche a livello di Sistema Operativo, i PLC sono caratterizzati dall'orientamento all'esecuzione di programmi di controllo. Infatti, i tipici compiti necessari per il controllo di un sistema automatizzato (**lettura "sincrona" del valore degli Input - elaborazione programma - attuazione "sincrona" del valore degli Output**) sono realizzati autonomamente in modo **ciclico**, così da permettere al programmatore di non doversi occupare delle problematiche di acquisizione dati e attuazione uscite. Poichè normalmente il tempo di campionamento ottenuto con questo funzionamento ciclico non è fissato in modo rigido (i.e. ciclo *libero*), per garantire un campionamento più stringente per alcuni gruppi di segnali il Sistema Operativo di un PLC fornisce delle minime funzionalità di **multi-tasking**, prevedendo normalmente uno o più (generalmente in numero limitato) processi periodici a priorità maggiore di quella del ciclo principale. Lo schema del ciclo di elaborazione del PLC, può essere quindi generalizzato come mostrato in figura B.3.6, nella quale si indica con:

- **IPI**: acquisizione della Immagine di Processo degli Ingressi
- **IPO**: attuazione dell'Immagine di Processo delle Uscite
- **PU**: Programma Utente
- **SO**: Sistema Operativo.

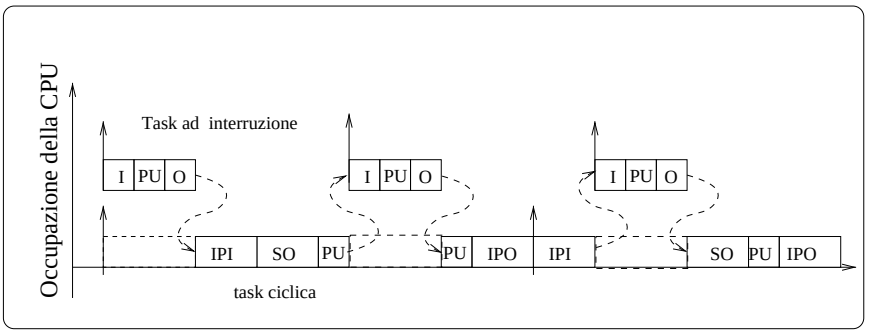


Figura B.3.6: Schema del ciclo di elaborazione del PLC.

B.3.2 Sistemi di controllo distribuiti

I sistemi di controllo distribuiti (**Distributed Control System, DCS**), sono sistemi utilizzati per controllare processi manufatturieri di grandi dimensioni che hanno la caratteristica di essere distribuiti geograficamente (es. un impianto di raffinazione petrolchimica, produzione energia elettrica, ecc.). Gli elementi che caratterizzano il DCS sono quindi la possibilità di poter collocare in diversi punti remoti i moduli per l'acquisizione dati, l'elaborazione e il controllo, con la conseguente necessità di avere una rete di comunicazione il più possibile efficiente che metta in collegamento fra di loro i vari sottosistemi.

Dal punto di vista hardware, gli elementi costitutivi di un DCS appartengono alle seguenti categorie:

1. **Stazioni di supervisione e configurazione**: sono tipicamente delle "workstation" o dei PC dotati di software specifici per:
   - interfaccia Uomo-Macchina (*Human-Machine Interface* o HMI), con pannelli di visualizzazione (quadri sinottici) dello stato dell'intero processo.
   - raccolta ed elaborazione statistica dei dati di processo (**SCADA**)
   - controllo dell'avanzamento delle varie macro-fasi del processo produttivo (**Batch Manager**)
   - Programmazione delle unità di controllo e misura remote (**Remote Terminal Unit, RTU**, v. punto successivo)
2. Le **Remote Terminal Units (RTU)**, che possono essere:
   - **Regolatori Analogici** che realizzano il controllo in retroazione di uno specifico microprocesso (es. temperatura/livello di un serbatoio)
   - **Controllori Logici** per il controllo della sequenza produttiva di una parte localizzata del processo.
   - **Moduli di I/O** non "intelligenti", eventualmente progettati in modo specifico per ambienti pericolosi.
3. **Sistemi di comunicazione**: tradizionalmente i segnali analogici dai sensori venivano inviati tramite loop di corrente $4 \div 20$ mA, ora i collegamenti vengono gestiti con **reti digitali**, che permettono una totale integrazione dei componenti del DCS.

Si noti che i controllori logici usati in un DCS possono essere PLC "tradizionali", ma più spesso sono dispositivi specifici o, eventualmente, controllori "ibridi", che integrano potenti processori orientati all'elaborazione numerica Floating-Point, in grado di unire le funzionalità di controllo logico a quelle di regolazione di sistemi continui. Inoltre, in genere tutti i componenti di un DCS sono in

grado di supportare le esigenze di ridondanza, vale a dire la contemporanea presenza di più dispositivi (processori, reti di comunicazione) che eseguono lo stesso compito, in modo che se uno di questi si dovesse guastare, un altro subentrerebbe senza problemi di sicurezza (sistema **fail-safe**). Un esempio dell'architettura completa di un DCS (Honeywell PlantScape) è schematizzato in Figura B.3.7.

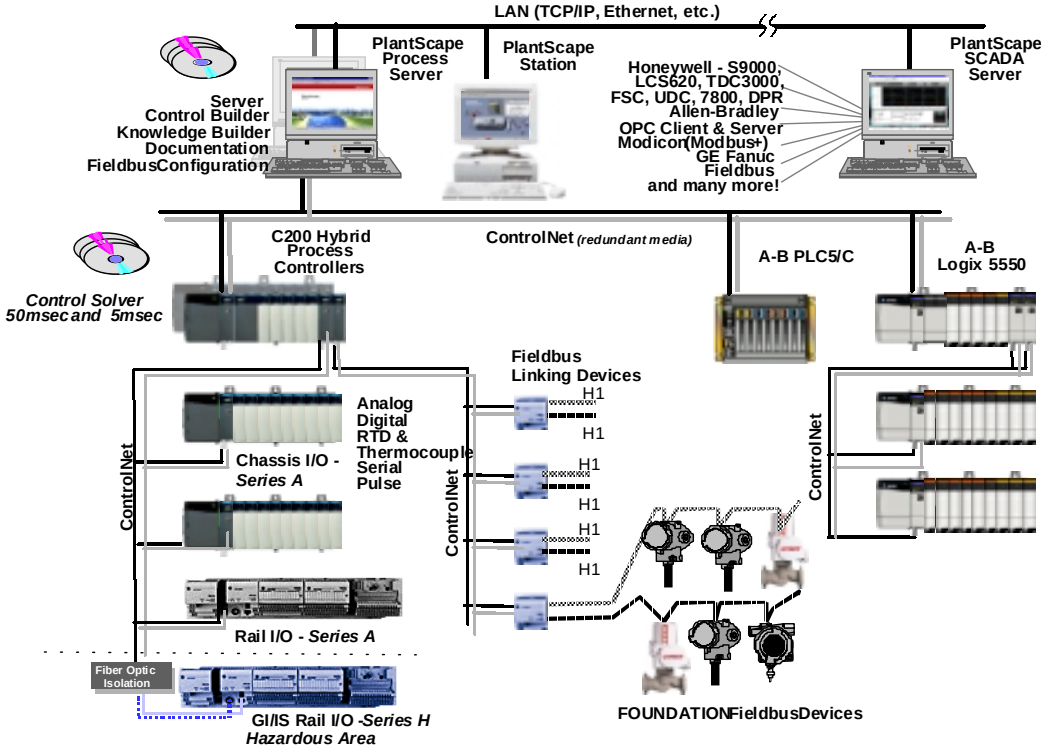


Figura B.3.7: Diagramma schematico dei componenti di un DCS (fonte Honeywell)

Anche dal punto di vista software, la programmazione dei DCS è studiata appositamente per poter governare un sistema di controllo con caratteristiche distribuite. In particolare, per astrarre completamente dal vincolo geografico a cui sono legati i moduli, ciascuno di questi (acquisizione, controllo, attuazione) è definito come proprietario di un certo **data-base** di informazioni. Le informazioni contenute in un certo modulo sono sempre visibili a livello globale (stazioni di supervisione e configurazione) e contrassegnate da un identificatore univoco chiamato **tagname**. Nel DCS ogni *tag* rappresenta una variabile di processo (input), una funzione o il valore di un'uscita. Ad esempio, il comando dell'apertura di una valvola può essere associato ad un tag chiamato `FV100.OPEN`. Il messaggio che prevede l'attuazione di quel comando viene inviato, tramite la rete, al modulo che è proprietario dell'output `FV100.OPEN`, che eseguirà il comando generando un messaggio di risposta.

B.3.2.1 Sistemi di Supervisione ed Acquisizione Dati (SCADA)

Una funzionalità molto importante per la gestione di impianti di grandi dimensioni è quella relativa all'acquisizione, visualizzazione ed elaborazione statistica dei dati di processo, realizzata da programmi software chiamati **Supervisory Control And Data Acquisition, SCADA**. Le caratteristiche di un sistema SCADA possono essere così riassunte:

- Presenta un'interfaccia HMI per gli operatori dell'impianto con pannelli di comando e quadri sinottici, che mostrano lo stato di funzionamento dell'intero sistema, consentono l'introduzione di parametri di lavorazione e la gestione di situazione di allarme.

- Acquisisce dati dal campo ad intervalli di campionamento abbastanza ampi (non esegue direttamente compiti di regolazione) e li memorizza in archivi permanenti, garantendone la protezione ed il salvataggio di *backup*.
- Crea e stampa rapporti cartacei e grafici per l'analisi statistica dei dati di produzione.
- Esegue elaborazioni quali il calcolo di medie e previsioni di andamento futuro, al fine di programmare interventi di manutenzione, correggere (adattare) i parametri di controllo, suggerire modifiche delle strategie produttive.

Le Figure B.3.8 e B.3.9, mostrano rispettivamente un possibile quadro sinottico e un grafico di analisi dei dati di un moderno sistema SCADA.

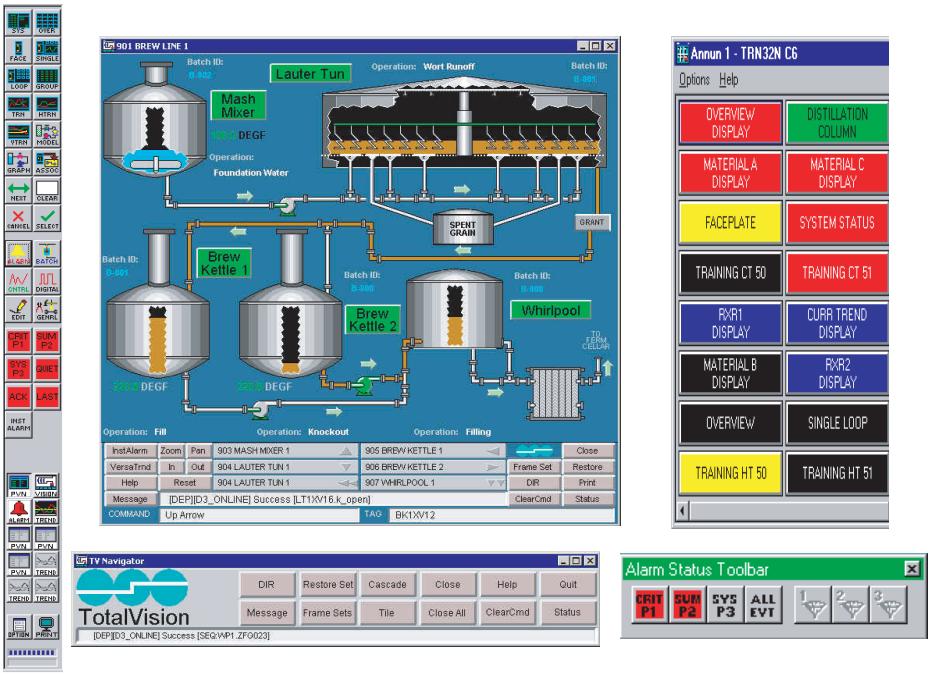


Figura B.3.8: Esempio di quadro sinottico di un generico sistema SCADA

B.3.3 Sistemi di comunicazione per architetture di controllo distribuite

I sistemi di controllo distribuiti necessitano "per natura" di sistemi di comunicazione, eventualmente di diversa tipologia, per l'integrazione dei componenti geograficamente distanti tra loro. Tuttavia, le esigenze di integrazione tra i vari livelli informativi delle aziende manufatturiere (**Computer Integrated Manufacturing, CIM**) e la necessità di ridurre le problematiche di cablaggio di sensori ed attuatori a bordo macchina, rendono le reti di comunicazione sempre più importanti anche per i sistemi di controllo di impianti di medie-piccole dimensioni. Le reti di comunicazioni utilizzate nell'automazione industriale, si possono distinguere in tre categorie (si veda anche Figura B.3.10):

- **Le Reti Informatiche**, che collegano i sistemi di supervisione di alto livello con altri sistemi informativi aziendali azienda. Su queste reti vengono scambiate grandi quantità di informazioni, relative ai dati di produzione dello stabilimento, alla gestione delle scorte di magazzino e della contabilità generale dell'azienda. La trasmissione di dati in questo tipo di reti non deve

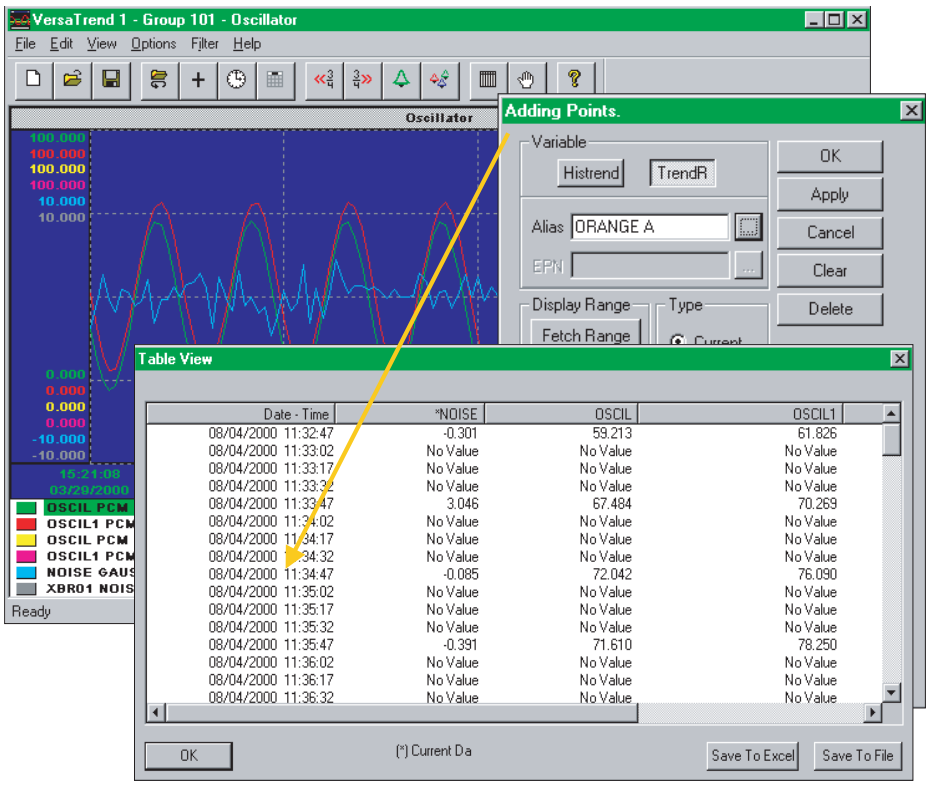


Figura B.3.9: Esempio di elaborazione statistica in un un generico sistema SCADA

soddisfare specifiche di tempo reale, e quindi possono essere utilizzati protocolli di comunicazione standard in modo da facilitare l'integrazione di differenti sistemi. L'esempio più tipico di protocollo per questo tipo di reti è il TCP/IP per la rete Ethernet.

- **Le Reti per il Controllo**, utilizzate per assicurare la comunicazione tra i dispositivi dedicati al controllo degli impianti. In questo caso le informazioni debbono essere scambiate in modo da:garantire l'assenza di errori e rispettare determinati vincoli temporali sui tempi di trasmissione dei dati e sull'occupazione della linea di comunicazione. Infatti, queste reti sono utilizzate per scambiare informazioni tra i sistemi di controllo (ad es. PLC) al fine di sincronizzare le operazioni tra sottoparti di un processo produttivo: per garantire la sincronizzazione degli eventi occorre che i ritardi nella trasmissione delle informazioni siano **contenuti** e **deterministici** (è sempre possibile prevederne l'entità).

- **Le Reti di Campo (Fieldbus)**, utilizzate per scambiare informazioni tra un sistema di controllo (ad es. PLC) e i sensori ed attuatori. Con una *rete di campo* i sensori e gli attuatori sono cablati ad un dispositivo di acquisizione "locale", fornito di un modulo di comunicazione e quindi collegati al sistema di controllo attraverso il cavo (unico) di una rete *fieldbus*. In alcuni casi, i sensori ed attuatori stessi sono dotati di interfaccia di comunicazione (sensori/attuatori "intelligenti"), per semplificare ulteriormente il cablaggio (es. encoder con interfaccia fieldbus, anzichè N fili; motori elettrici con azionamento integrato, ecc.). Chiaramente, i vincoli temporali richiesti ad un fieldbus sono legati all'esecuzione ciclica del programma del sistema di controllo: il tempo richiesto per acquisire gli ingressi "remoti" e attuare le uscite "remote" non deve influire eccessivamente sul tempo di ciclo totale del programma. Anche per i fieldbus, quindi, sono necessarie caratteristiche di determinismo nella trasmissione analoghe a quelle richieste alle reti per il controllo (livello superiore), con vincoli ancora più stringenti.

  Tra i vantaggi ottenibili con la Rete di Campo:

– La semplificazione architetturale, in quanto le reti sono facilmente espandibili e riconfigurabili.
– La riduzione del cablaggio, con conseguente diminuzione dei costi di installazione.
– La possibilità di trasmettere informazioni di alto livello, come ad esempio funzioni di autodiagnosi.
– La possibilità di calibrare e configurare i parametri (es. soglia di attivazione di un sensore logico, ecc.) dei sensori/attuatori intelligenti via software, da un unico terminale connesso alla rete.
– Una maggiore robustezza delle informazioni, in quanto la trasmissione digitale è meno sensibile ai disturbi.

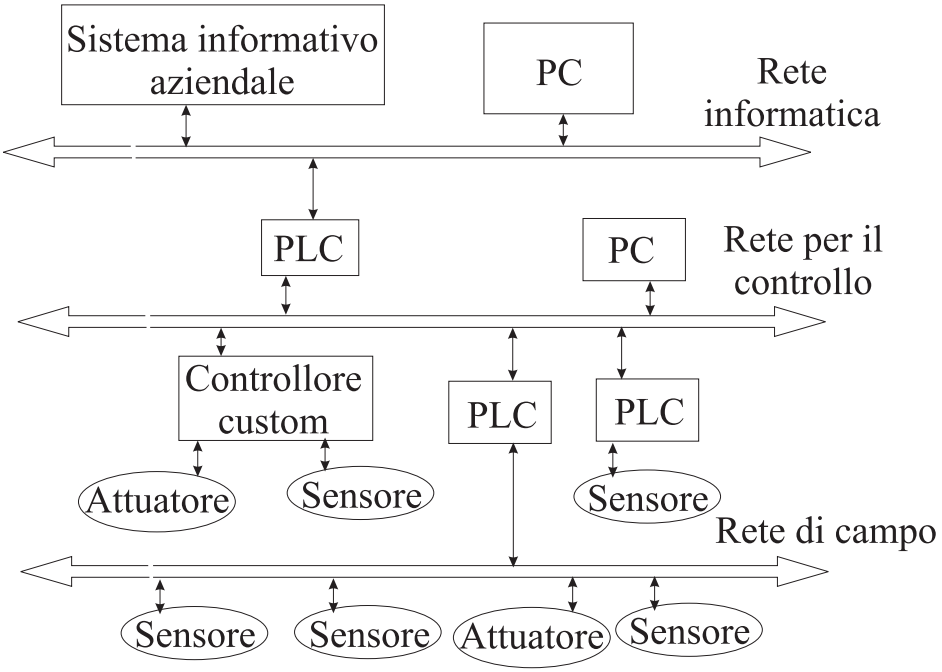


Figura B.3.10: Reti per l'automazione

Si noti che la quantità di informazioni scambiate è maggiore al livello più "alto" della gerarchia delle reti di comunicazione per l'automazione, mentre le esigenze di determismo sono maggiori al livello più "basso" (fieldbus).

B.3.3.1 Generalità sulle reti di comunicazione digitale

Le reti di comunicazione digitali si possono classificare secondo differenti modalità, a partire dalla tipologia di connessione fisica dei nodi della rete, dall'architettura fisica di tali collegamenti fino ad arrivare alla modalità logica di invio e riconoscimento delle informazioni. La definizione delle specifiche di tutte queste caratteristiche, determina il **protocollo** che deve essere rispettato da un dispositivo per interfacciarsi ad una data rete. Per stabilire un metodo comune di definizione dei protocolli di rete, l'organismo internazionale **ISO**, ha stabilito, nel 1978, un modello concettuale standardizzato dei servizi di comunicazione, denominato **OSI (Open System Model)**.

Il modello ISO/OSI definisce un insieme di funzionalità che devono essere realizzate da ogni nodo di una rete digitale, organizzandole secondo una gerarchia a 7 livelli. Ogni livello funzionale della gerarchia ISO/OSI è "costruito" sulla base delle funzionalità già eseguite dai livelli inferiori. Nella definizione di un protocollo per una determinata rete di comunicazione, soprattutto quando esso

sia pubblicato come standard internazionale, occorre quindi specificare per ogni livello ISO/OSI la modalità di realizzazione delle funzionalità di quel livello.

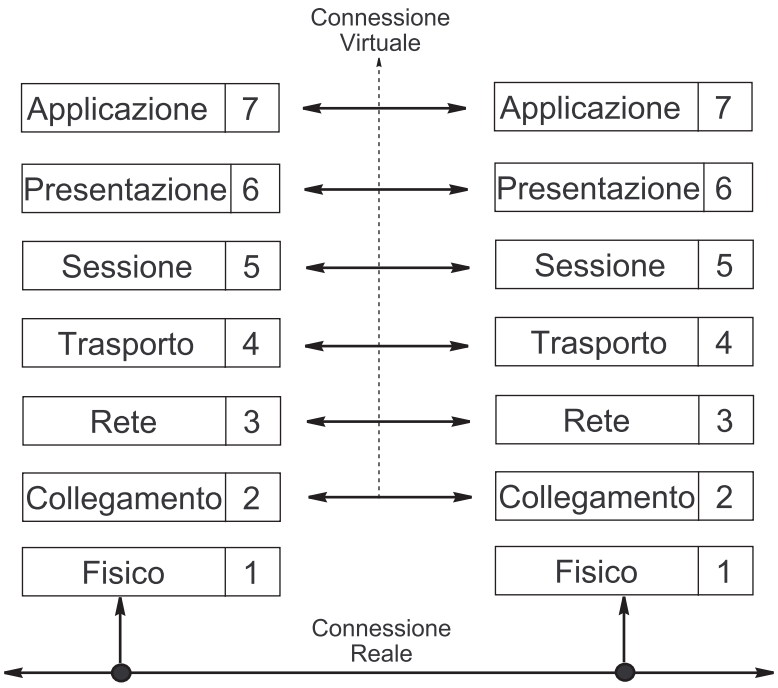


Figura B.3.11: Livelli funzionali del modello ISO/OSI per reti di comunicazione

I livelli del modello ISO/OSI, schematicamente rappresentati in Figura B.3.11 sono, partendo dal livello 1 fino al livello 7:

1. **Fisico**: definisce i collegamenti dal punto di vista elettrico e meccanico (**hardware**), le velocità e le modalità di trasmissione dati.

2. **Collegamento**: realizzato, come i successivi, da procedure **software**, definisce come verificare la correttezza dei bit inviati e come riconoscere i destinatari dei dati trasmessi, organizzandoli in strutture standard dette **frame**.

3. **Rete**: garantisce un collegamento logico anche tra nodi non fisicamente connessi, svolgendo in sostanza funzioni di instradamento dei dati.

4. **Trasporto**: assicura l'integrità dei messaggi composti da molteplici sequenze di bit ("frammentazione" e "ricomposizione"). Rappresenta l'interfaccia tra le funzionalità di rete vere e proprie e quelle del software applicativo (realizzate dai tre livelli successivi).

5. **Sessione**: gestisce le funzioni di sincronizzazione dei nodi in caso di errore, permette l'identificazione dei dispositivi collegati (autenticazione accessi) ed altri servizi necessari al controllo del dialogo fra i nodi.

6. **Presentazione**: si occupa delle problematiche di interpretazione delle informazioni che sorgono quando si desidera realizzare una struttura distribuita. Ad esempio, a questo livello sono realizzate le funzioni di associazione tra stringhe e sequenze di bit, di compressione dei dati, di crittografia, ecc.

7. **Applicazione**: fornisce interfacce e servizi ai programmi applicativi per rendere trasparente l'accesso alle risorse remote tramite la rete digitale. Ad esempio, servizi di livello 7 sono il trasferimento di interi file, la possibilità di operare su data-base distribuiti ecc.

Un protocollo di rete non deve necessariamente specificare le funzionalità per ogni livello del modello ISO/OSI. Infatti, specificare ogni livello significa appesantire il controllo necessario per la comunicazione, pertanto anche la capacità computazionale da dedicare ad essa. Negli standard internazionali che definiscono protocolli per fieldbus, ad esempio, sono normalmente definiti solamente i **primi due livelli**, per alleggerire i compiti di ciascun nodo della rete. Un eventuale protocollo di settimo livello (*applicazione*), che in questo contesto definisce servizi per programmi di controllo, permette di definire una rete di controllori sulla base di un bus di campo. La maggiore semplicità dei protocolli per reti di campo deriva dal fatto che esse sono maggiormente orientate alla possibilità di comunicare con dispositivi di "intelligenza" limitata (semplicità), ed alla necessità di garantire rapidità e determinismo nell'invio/ricezione e riconoscimento dati (Real-Time).

Tra le caratteristiche che devono essere specificate per il livello Fisico del modello ISO/OSI, vi sono:

- i **livelli di tensione** (o corrente) ammessi per rappresentare i valori logici
- le **tolleranze massime** sui valori logici e sui tempi di salita, commutazione ecc. dei segnali elettrici
- il **tipo di codifica** di un singolo bit.
- la **modalità di trasmissione**, sincrona o asincrona, half-duplex o full-duplex.
- il tipo di **controlli di integrità dei bit** che devono essere eseguiti per riconoscere ed eliminare errori nella trasmissione dei dati.
- le **caratteristiche fisiche** dei cavi e dei connettori e la topologia di connessione dei nodi.

Per la trasmissione fisica di dati, i segnali elettrici vengono solitamente inviati e ricevuti tramite opportuni cavi, le cui caratteristiche sono anch'esse specificate nella definizione di un protocollo di rete. Tra le tipologie di cablaggio più diffuse nell'automazione, si possono citare:

- il **doppino intrecciato**: il più semplice mezzo trasmissivo di segnali logici è un collegamento elettrico in rame tramite un cavo bipolare, i cui fili vengono normalmente intrecciati per bilanciare l'esposizione ai disturbi elettromagnetici, ed eventualmente schermati con una guaina metallica.
- i **cavi coassiali**: in questo caso i poli sono fisicamente concentrici. Presentano normalmente una maggiore immunità ai disturbi, ma anche una maggiore rigidezza e difficoltà di cablaggio.
- le **fibre ottiche**: permettono le massime prestazioni in termini di larghezza di banda e immunità ai disturbi. D'altra parte sono costose e difficili da installare, dato che richiedono particolari accorgimenti nelle operazioni di taglio.

Infine, un importante servizio fornito nel livello 1 del modello ISO/OSI è il controllo dell'integrità delle informazioni ricevute. Un meccanismo molto semplice di controllo è l'inserimento di un bit ridondante per ogni byte di dato che renda pari (o dispari) il numero di bit a "1" inviati (**bit di Parità**). In questo caso, si può parlare di controllo "orientato al carattere" (byte). Tuttavia, nel caso in cui vi siano 2 bit sbagliati nel dato, tale meccanismo non è più sufficiente a riconoscere il byte come sbagliato. In generale, il numero di bit errati che non è possibile riconoscere con una certa modalità di controllo viene detto **distanza di hamming**. Distanze di hamming superiori a 2 si possono ottenere con il meccanismo detto **Cyclic Redundancy Check (CRC)**, con il quale si inviano dei byte aggiuntivi il cui valore dipende dalla sequenza di bit di dato inviati in precedenza, secondo un certo **polinomio generatore**. Il CRC viene ri-calcolato dal ricevente sui bit di dato ricevuto, e se non coincide con il CRC inviato si notifica l'errore.

Si noti che, in genere, il controllo tramite CRC avviene inserendo opportuni campi all'interno dei "pacchetti" (*frame*), inviati. Pertanto, questa funzionalità viene solitamente inserita fra quelle di livello 2 (Collegamento). Infatti, le funzionalità di livello Collegamento si possono suddividere in:

- Controllo dell'accesso al mezzo fisico (**Medium Access Control o MAC**), secondo una determinata politica di allocazione ed un protocollo di richiesta.

- Preparazione del "frame" di dati, inserendo le informazioni necessarie alla realizzazione di un collegamento logico tra i nodi, e riconoscimento del nodo come destinatario dei dati. (**Logic Link Control o LLC**).

Il protocollo di accesso al mezzo (MAC) può essere di tipo **deterministico** o **controllato**. Nel primo caso, la risorsa fisica viene ripartita equamente tra tutti i nodi della rete su base temporale o frequenziale (**Time Division Multiple Access, TDMA, o Frequency Division Multiple Access, FDMA**): ogni nodo può trasmettere dati nel proprio intervallo di tempo o sulla portante alla propria frequenza. Nel caso di accesso controllato, la risorsa fisica non viene allocata rigidamente, ma in modo dipendente dall'effettiva necessità di trasmettere (o ricevere) di ogni nodo, per avere maggiore flessibilità ed espandibilità. Possibili protocolli di accesso controllato sono:

- **Master/Slave**, o arbitrato: esiste un solo arbitro della rete (il "master") che decide quale nodo può trasmettere, interrogando periodicamente ciascun nodo sulla necessità di trasmettere ("polling"). Il grande vantaggio di questa modalità è l'assoluto determinismo nel calcolo dei ritardi di trasmissione (un ciclo di polling). D'altra parte, richiede che un nodo della rete, il master, sia più complesso da programmare e configurare degli altri, gli slave.

- **Multi-Master**: ogni nodo può decidere quando trasmettere, interrogando solamente lo stato del mezzo fisico (libero o occupato). Occorre risolvere le problematiche relative a possibili trasmissioni contemporanee ("collisioni") sul canale, che corrompono la validità dei dati. Una possibile tecnica, adottata nella rete Ethernet, è detta CSMA/CD (Carrier Sense Multiple Access/Collision Detect): i nodi che trasmettono sono in grado di rilevare il conflitto e di decidere quando ritrasmettere il dato senza rischiare una nuova collisione. I vantaggi sono legati alla possibilità di trasmissione immediata, senza ritardi, nel caso più "fortunato". D'altra parte, se il traffico sulla rete è notevole, vi saranno molteplici collisioni e quindi i ritardi sono fortemente **non deterministici**.

- **Token Passing**: il token (o gettone) rappresenta il diritto di trasmettere e corrisponde ad uno specifico tipo di pacchetto dati che circola sulla rete. Ogni nodo che debba trasmettere si mette in attesa che il token sia disponibile sulla rete, prelevandolo, oppure gli sia inviato esplicitamente. Al termine della trasmissione, effettuerà il rilascio del token inviandolo come fosse un qualunque altro dato. Il vantaggio del Token Passing è che il tempo di attesa ha un valore massimo corrispondente al caso peggiore (attraversamento completo della rete da parte del token ed ogni nodo con necessità di trasmettere).

Infine, per la parte Logical Link Control (LLC) del livello 2, vi sono in genere due modelli di comunicazione: **Source/Destination** o **Producer/Consumer**. Nel modello Source/Destination il frame contiene esplicitamente gli indirizzi del nodo sorgente e del nodo destinazione del dato. In questo modo si possono realizzare collegamenti virtualmente "punto a punto", tra nodi anche non direttamente connessi dal mezzo fisico (es: punti a grande distanza su un bus lineare). Nel modello Producer/Consumer, invece, il frame contiene un identificatore che è però relativo alla tipologia di contenuto del messaggio. In questo modo vi possono essere molti nodi "consumatori" interessati al dato e che possono prelevarlo contemporaneamente, evitando che ogni nodo debba essere destinatario di un diverso messaggio contenente lo stesso dato. Il vantaggio di quest'ultima metodologia è certamente uno sfruttamento più efficiente della risorsa fisica.

Per quanto riguarda i livelli superiori al secondo del modello ISO/OSI, essi, come detto, non sono in genere considerati nella definizione dei protocolli per fieldbus, in modo da definire reti "snelle" e di semplice realizzazione. D'altra parte, molto spesso il fieldbus viene anche usato per fare comunicare dispositivi di controllo di pari livello nella gerarchia del sistema automatizzato (*rete per il controllo*). Per tale scopo vengono definiti protocolli specifico di **livello applicativo** (livello 7 ISO/OSI), secondo standard che ammettono l'uso di determinati fieldbus (anche più di uno) per i livelli 1 e 2.

Tra gli innumerevoli fieldbus per applicazioni industriali, si citano qui a puro titolo di esempio:

- **CAN (Controller Area Network)**: bus nato nell'ambito dell'industria automobilistica, con esigenze di costo contenuto, elevata immunità ai disturbi, elevata capacità di rilevazione e correzione di errori. Al di sopra dei servizi di livello 1 e 2 definiti nello standard CAN, esistono diversi standard che definiscono protocolli di livello 7. Fra questi, i più noti sono:

  - **CANopen**: protocollo di livello *applicazione* per il bus CAN, progettato per semplificarne le applicazioni a livello industriale come Rete di Controllo.
  - **DeviceNet**: quest'ultimo è molto utilizzato nell'automazione grazie al fatto di essere diventato il bus di riferimento di Rockwell Automation, il maggiore produttore americano di controllori per l'industria.

- **FOUNDATION Fieldbus**: standard molto diffuso nel controllo di processi con DCS, in quanto nato espressamente per sostituire il loop di corrente analogico $4 \div 20$mA, con una rete digitale altrettanto robusta e idonea ad ambienti pericolosi.

- **PROFIBUS (PROcess FIeldBUS)**: standard molto diffuso in Europa (soprattutto perchè supportato "ufficialmente" dal gruppo Siemens), ne esistono 3 diverse sotto-definizioni, per una delle quali (Profibus-FMS) è definito un protocollo di livello 7 simile allo standard IEC 9506-5 **Manufacturing Message Specification (MMS)**.

In generale, i vari fieldbus definiscono caratteristiche elettriche e di cablaggio molto differenti. Tuttavia, è necessario notare che la diffusione di massa delle tecnologie per la rete Ethernet ha stimolato l'interesse per tale standard anche nel campo industriale. L'estensione della rete Ethernet come fieldbus o rete di controllo è l'oggetto di numerosi standard relativamente recenti, che si distinguono soprattutto per la filosofia con cui affrontano le esigenze di determinismo e affidabilità che tipicamente Ethernet per uso "office" non è in grado di soddisfare. Tra questi protocolli per automazione su base Ethernet, si possono citare:

1. Quelli basati su **incapsulamento dei dati in pacchetti TCP o UDP** del protocollo TCP/IP. Tale soluzione non garantisce il determinismo delle trasmissioni, se non sfruttando opportuni moduli di **routing** intelligenti, ma permette di usare hardware e software standard e a basso costo. Adottano questa strategia:

   - **Modbus/TCP**: estensione del protocollo **Modbus/RTU** che definisce uno specifico (e comune a Modbus/TCP) livello 7 e adotta RS-232 o RS-485 come livello fisico, ovviamente sostuito da connessioni TCP/IP su base Ethernet.
   - **Ethernet/IP (Industrial Protocol)**: adatta il livello 7 della reti DeviceNet (su base CAN) e ControlNet (su protocolli di livello inferiore proprietari di Rockwell Automation) alle connessioni TCP/IP.
   - **ProfiNet**: adatta i concetti di Profibus allo strato TCP/IP.

2. Quelli che ridefiniscono le metodologie di accesso, trasformando di fatto Ethernet **da rete multi-master a rete master-slave**. Ad esempio:

   - **Ethernet PowerLink (EPL)** suddivide i nodi in **Managing Node (MN)** (unico) e **Controlled Node (CN)**. Il nodo MN è l'arbitro di tutte le trasmissione e stabilisce un ciclo di traffico sincrono a ripartizione degli slot temporali (politica TDMA). Il pacchetto base Ethernet non è però modificato ed il protocollo EPL può essere implementato interamente via software, usando schede di rete Ethernet standard.

3. Quelli che ridefiniscono completamente **l'hardware di gestione del bus Ethernet**. Ad esempio:

   - **EtherCAT (Control Automation Technology)** suddivide i nodi in **master** e **slave**, tra i quali viene trasmesso un unico pacchetto Ethernet modificato ***on-the-fly*** durante il passaggio da uno slave al successivo in una topologia ad anello. Questa soluzione minimizza i tempi di ciclo e le performance complessive della rete, tanto che EtherCAT risulta attualmente molto usata anche per le applicazioni di Motion Control. Tuttavia, l'implementazione di uno slave EtherCAT non può essere fatta con schede di rete Ethernet standard, ma richiede hardware specifico la qui proprietà intellettuale è riservata dall'azienda Beckhoff Automation.

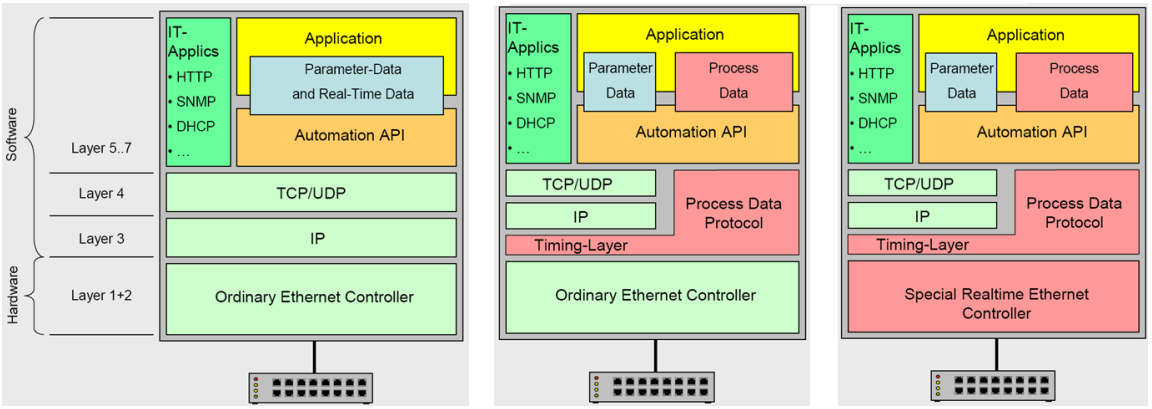


Figura B.3.12: Strategie per l'uso di Ethernet come fieldbus: 1) incapsulamento dati in pacchetti TCP/UDP standard (a sinistra); 2) ridefinizione della politica multi-master in master-slave (al centro); 3) ridefinzione dell'hardware di gestione del bus (a destra)

B.3.3.2 Il Bus di Campo CAN

Il bus CAN fu introdotto nella metà degli anni '80 dalla Bosch, per risolvere i problemi di cablaggio fra sensori e centraline a microcontrollore nell'industria automobilistica. Si è poi diffuso anche in altri settori (aerospaziale, automazione, ecc.) grazie alla semplicità ed ai costi di implementazione contenuti. Sebbene sviluppato inizialmente da Bosch, il protocollo è regolamentato dallo Standard ISO 11898, e si è poi rivelato particolarmente adatto per la realizzazione di semplici sistemi di controllo con intelligenza distribuita (microcontrollori).

Dal punto di vista del livello Fisico, l'architettura della rete CAN è a Bus Condiviso, cioè i messaggi trasmessi sono potenzialmente ricevuti da tutti i nodi connessi. Benchè non sia esplicitamente definito nello standard, il mezzo trasmissivo più usato è un semplice doppino intrecciato, che deve essere dotato di resistenze di terminazione ai capi liberi del cavo (per l'**adattamento** della linea) con valore tipico di 120Ω, come indicato in Figura B.3.13.

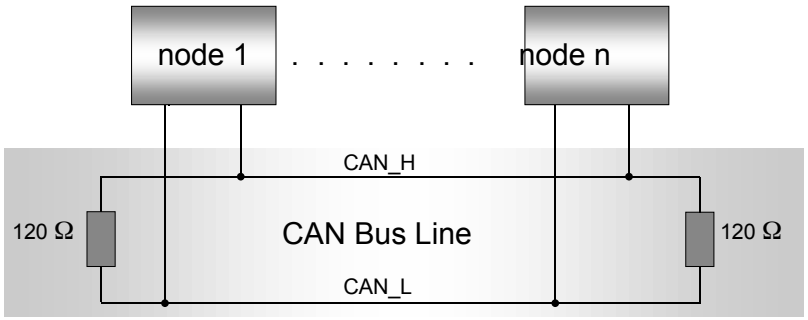


Figura B.3.13: Architettura fisica tipica del bus CAN

La tecnica di segnalazione è di tipo **differenziale sincrona**, con il valore logico "0" **dominante** (cioè prevalente in caso di trasmissione contemporanea) e valore "1" **recessivo**. Il valore "0" corrisponde ad una differenza di potenziale tra **CAN-H** e **CAN-L** superiore a 0.9 V, ed il valore "1" ad una differenza inferiore a 0.5 V. Nominalmente, CAN-H e CAN-L in configurazione "0" sono rispettivamente a 3.5 e 1.5 V. Le codifiche dei bit ammesse sono NRZ (più usata) e Manchester, e la velocità di trasmissione è compresa tra 20 Kbit/s e 1 Mbit/s.

Dal punto di vista dell'accesso al mezzo fisico (Medium Access Control, MAC), il Bus CAN è una rete **multi-master** ad accesso casuale (**Carrier Sense Multiple Access, CSMA**), simile in

Velocità di trasmissione (Kbit/s)	Estensione della rete (m)
1000	40
800	50
500	100
250	250
125	500
50	1000
20	2500

Tabella B.3.1: Valori ammessi dallo standard CAN

questo alla rete Ethernet. Tuttavia, a differenza di quest'ultima, prevede un meccanismo di gestione delle collisioni **non distruttivo** per i dati trasmessi (**Collision Avoidance, CSMA/CA**), basato sulla priorità dei messaggi. Ogni trasmissione inizia con una fase di arbitraggio che permette di risolvere i conflitti sul bus prima che tutti i dati trasmessi siano corrotti. In sostanza, la fase di arbitraggio prevede che ogni nodo rilevi lo stato del canale, mentre sta trasmettendo il primo campo utile del messaggio, che è **l'identificatore** del tipo di dato. Se rileva uno stato dominante mentre sta trasmettendo un bit recessivo, **perde la contesa** e si predispone nuovamente in ricezione ("listening"), come mostrato in Figura B.3.14. Ovviamente, perchè il meccanismo di arbitraggio funzioni correttamente, occorre che non esistano identificatori di messaggio identici per più nodi. I bit dell'identificatore rappresentano infatti il livello di **priorità del messaggio**, che viene definita in modo statico in fase di configurazione della rete. Inoltre, per poter distinguere messaggi provenienti da nodi diversi, solitamente una parte dell'identificatore viene associata al nodo che trasmette il pacchetto. Tuttavia, su quest'ultimo punto (identificazione e assegnazione di priorità fra i nodi) il protocollo CAN non specifica nulla, cosicchè solitamente queste problematiche vengono risolte a livello di protocollo *applicazione*.

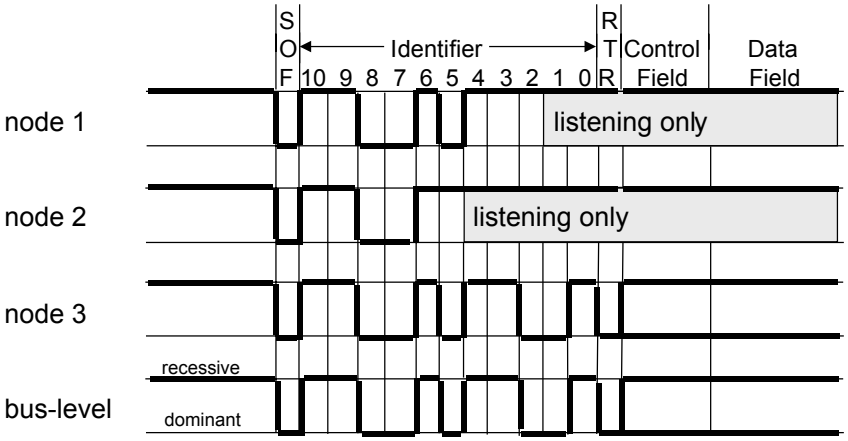


Figura B.3.14: Esempio di arbitraggio con protocollo CAN: il nodo 3 vince la contesa e guadagna il diritto a continuare la trasmissione

Il metodo di indirizzamento CAN (parte Logical Link Control, LLC, del protocollo di livello 2) è del tipo **producer/consumer**, cioè il frame contiene un identificatore riferito al contenuto del messaggio, piuttosto che all'indirizzo di un singolo nodo destinatario. In base all'identificatore, ogni nodo deciderà poi se acquisire o meno il dato. I servizi offerti dall'LLC del CAN sono denominati:

- **L_DATA**: un dato viene trasmesso dal nodo e messo a disposizione di tutti i nodi interessati.
- **L_REMOTE**: il nodo invia una richiesta di trasmissione per un particolare dato. Il nodo interessato alla richiesta invierà poi il dato (a tutti) con L_DATA.

Il meccanismo di controllo dell'identificatore viene normalmente realizzato via hardware da un apposito circuito **controllore CAN**, il quale memorizza tale campo in un **registro di accettazione** e lo confronta con delle maschere di bit opportune. In questo modo, è possibile realizzare totalmente via hardware la gestione del protocollo. Questa possibilità è ampiamente sfruttata dai costruttori di microcontrollori e DSP orientati al controllo, i quali possono integrare con molta facilità il CAN controller e estendere così le potenzialità di comunicazione.

Il meccanismo di controllo degli errori nel CAN è basato su un campo CRC di 15 bit all'interno del frame, che garantisce una **distanza di hamming pari a 6**.

Come evidenziato in Figura B.3.15, il pacchetto trasmesso secondo il protocollo CAN può essere di due lunghezze massime differenti (CAN versione 2.0A e 2.0B, o formato esteso). Inoltre, sono ammesse quattro diverse tipologie di pacchetto, con funzionalità differenti:

- **Data Frame**, per implementare il servizio L_DATA
- **Remote Frame**, per inviare la richiesta di servizio di tipo L_REMOTE: il campo dati contiene l'informazione sul dato richiesto, che verrà poi inviato dal nodo ricevente con un pacchetto *Data Frame*
- **Error Frame**, per segnalare ricezione di messaggi errati
- **Overload Frame**, che sono messaggi "vuoti" (cioè senza un campo dati significativo), inviati da un determinato nodo per segnalare la propria necessità di rallentare gli scambi sulla rete in situazioni di "congestione".

I bit RTR, SRR e IDE servono appunto per distinguere il tipo di frame e se il formato è da intendersi **base** (2.0A) o **esteso** (2.0B, con identificatori più lunghi).

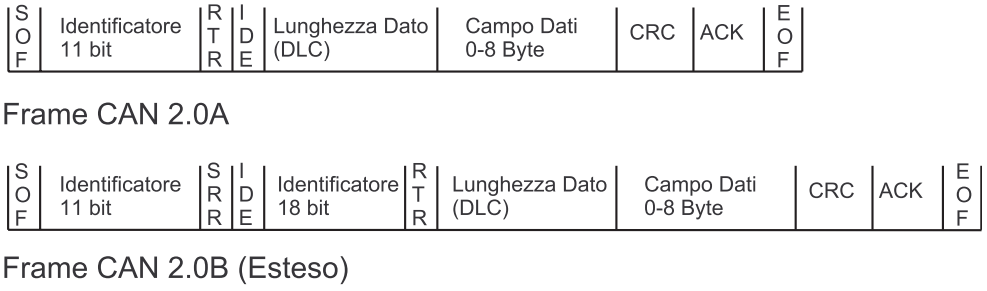


Figura B.3.15: Struttura dei pacchetti secondo il protocollo CAN (2.0A e 2.0B)

Si noti che la dimensione massima del campo dati "utili" in un frame CAN è di 8 byte. Rispetto ad altre reti di campo, questo valore risulta abbastanza limitato. Pertanto, l'efficienza nel trasferimento di grandi quantità di dati, risulta limitata, in quanto i dati devono essere "spezzettati" su molti pacchetti, per ognuno dei quali sarà richiesta la fase di arbitraggio. D'altra parte, se le dimensioni delle informazioni sono contenute e possono essere trasmesse con un solo frame, quest'ultimo può essere trasmesso potenzialmente in modo immediato, soprattutto in caso di priorità elevata del messaggio. Questo fa del CAN una rete molto orientata alla connessione fra dispositivi che richedono un traffico di limitate dimensioni, ma con spiccata capacità di immediata reazione ad "eventi". Risulta meno efficiente, qualora sia richiesta l'acquisizione ciclica di dati, ad esempio provenienti da moduli di I/O remoti, da parte di un controllore principale.

Protocollo di livello Applicazione per bus CAN: DeviceNet

Come detto, il tipico utilizzo di un fieldbus nell'automazione industriale, prevede proprio la connessione fra un dispositivo di controllo "master" e dei moduli remoti ("slave") che effettuano il solo interfacciamento "locale" agli I/O del campo. Per poter utilizzare il bus CAN a tale scopo, occorre realizzare dei meccanismi per una corretta assegnazione degli identificatori e, di conseguenza, delle

priorità fra i nodi e fra i tipi di messaggio. Le funzionalità principali dei protocolli di livello superiore basati sul CAN sono proprio orientate a risolvere queste problematiche. Ad esempio, il protocollo DeviceNet è stato studiato al fine di rendere il bus CAN idoneo ad applicazioni di controllo distribuito in ambiente industriale. Per semplificare le connessioni, definisce un **set rigido di identificatori** per un massimo di 64 nodi collegabili alla rete, suddividendo i possibili messaggi in 4 gruppi, come evidenziato in Figura B.3.16.

IDENTIFIER BITS											HEX RANGE	IDENTITY USAGE
10	9	8	7	6	5	4	3	2	1	0		
0	Group 1 Message ID				Source MAC ID						000 - 3ff	Message Group 1
1	0	MAC ID						Group 2 Message ID			400 - 5ff	Message Group 2
1	1	Group 3 Message ID			Source MAC ID						600 - 7bf	Message Group 3
1	1	1	1	1	Group 4 Message ID (0 - 2f)						7c0 - 7ef	Message Group 4
1	1	1	1	1	1	1	X	X	X	X	7f0 - 7ff	Invalid CAN Identifiers
10	9	8	7	6	5	4	3	2	1	0		

Figura B.3.16: Identificatori per pacchetti CAN ammessi dal protocollo DeviceNet

Tale set di identificatori, permette inoltre la realizzazione di comunicazioni di tipo "master/slave" necessarie per lo scambio dati, ad esempio, tra un PLC e un'unità di I/O remota, sulla base del protocollo (multi-master) CAN. I possibili canali "virtuali" di comunicazione possono essere:

- **Explicit messaging**, cioè scambio di *messaggi diretti espliciti*, il cui significato dipende dall'applicazione specifica (es. richiesta di un servizio ad un sistema remoto)
- **I/O Messaging**: cioè instaurazione di un rapporto **Master/Slave** per lo scambio di dati di I/O (sensori, attuatori). In questo caso, il Master può:
  - richiedere esplicitamente allo Slave un aggiornamento degli I/O
  - attivare uno scambio dati ciclico con lo Slave
  - attivare uno scambio dati in modo che essi vengano trasmessi solo in caso di "cambiamento di stato" di un segnale di I/O (**Change of State, CoS**)

Si noti che la modalità di trasmissione *Change of State* risulta particolarmente efficiente in caso di scarsa "dinamicità" dei sensori o attuatori utilizzati, mentre diventa controproducente nel caso in cui alcuni nodi siano troppo spesso sollecitati da cambiamenti di stato dei propri I/O.

B.3.3.3 Il Bus di Campo Profibus

La rete Profibus nasce espressamente per lo scambio di dati tra dispositivi remoti in un sistema di automazione, con modalità del tipo "master/slave" tipicamente richieste in tali applicazioni. Lo standard Profibus raggruppa tre possibili sotto-definizioni, ciascuna con un differente campo di applicazioni:

- **Profibus-DP** (Decentralized Periphery): basato sul livello fisico della linea seriale RS-485 (o fibra ottica), definisce oltre a questo il solo livello 2. È quindi molto idoneo per comunicazioni tra un controllore (PLC) e moduli remoti di I/O.

- **Profibus-PA** (Process Automation): differisce dal DP solo nel livello fisico, adatto a lavorare in ambienti potenzialmente esplosivi.
- **Profibus-FMS** (Fieldbus Message Specification): definisce un protocollo di livello 7 sulla base delle specifiche per Profibus-DP. Il protocollo FMS, leggera variante dello standard MMS, è studiato per la realizzazione di una Rete di Controllo.

Il livello fisico Profibus-DP può essere basato sulle modalità di trasmissione seriale derivate dallo Standard RS-485 (in modalità half-duplex con codifica NRZ), oppure su una connessione a fibre ottiche. Nel caso più diffuso della connessione RS-485, il mezzo trasmissivo è un doppino intrecciato (opzionalmente schermato) i cui poli A e B vengono detti **Tx/Rx-P** e **Tx/Rx-N**. Il bus è terminato con una resistenza tipica di 220Ω tra A e B e di 390Ω tra i poli e l'alimentazione, come evidenziato in Figura B.3.17.

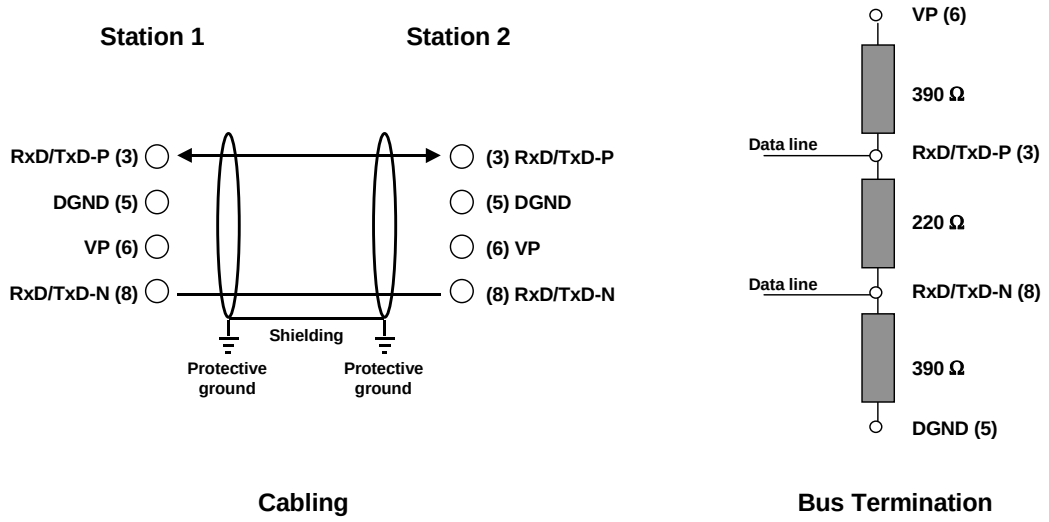


Figura B.3.17: Collegamento con linea RS-485 per la rete Profibus

Il valore logico "0" è codificato da una differenza di potenziale tra i due poli inferiore a 0.4V, mentre il valore logico "1" è codificato da una differenza di potenziale massima di 12V. I dati sono inviati secondo una modalità **asincrona**, cioè con un bit di start, uno di stop ed uno di parità per ogni byte di dato. Il numero massimo di nodi connessi ad un segmento RS-485 è 32, oltre i quali serve un ripetitore. La velocità di trasmissione ammessa è compresa tra i 9.6 Kbit/s e i 12 Mbit/s.

Velocità di trasmissione (Kbit/s)	Lunghezza del segmento RS-485 (m)
12000	100
1500	200
500	400
187.5	1000
19.2	1200
9.6	1200

Tabella B.3.2: Valori ammessi dallo Standard Profibus

Il livello fisico del Profibus-PA è anch'esso derivato da un altro standard, in particolare corrisponde a quello definito nello standard IEC 1158-2, che consente una maggiore sicurezza dei cavi e degli apparati, oltre alla possibilità di trasmettere l'alimentazione alle apparecchiature direttamente tramite il cavo di bus. Il segnale è trasmesso sotto forma di corrente, compresa tra ±9mA, con una codifica Manchester Il numero massimo di nodi sul segmento PA è 32, e la velocità massima di trasmissione è 32.125 Kbit/s.

Il protocollo di livello 2 del Profibus, denominato nello standard anche **Fieldbus Data Link (FDL)**, permette la realizzazione di comunicazioni sia **Master/Slave** che **Token Passing**, con un modello di indirizzamento dei nodi di tipo **Source/Destination**. I nodi presenti nella rete possono essere classificati come *master* o *slave*, come mostrato in Figura B.3.18.

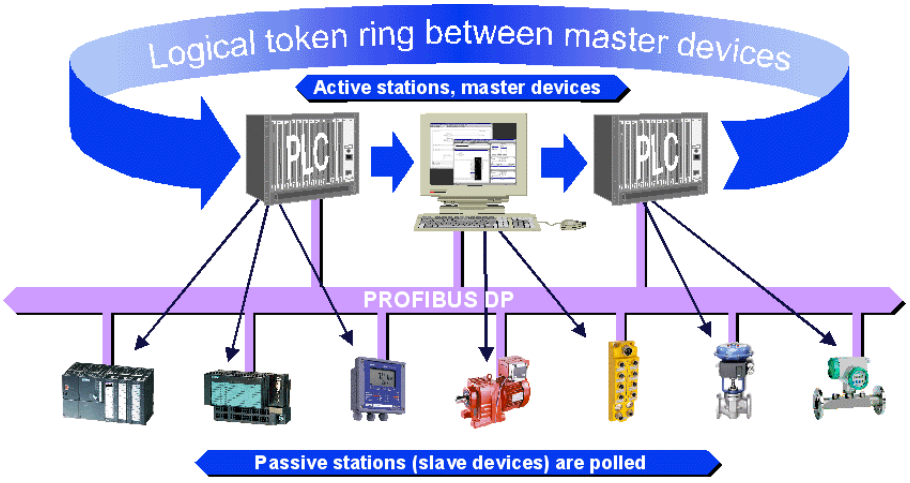


Figura B.3.18: Tipologia dei nodi Profibus e loro organizzazione logica

Ciascuno *slave* deve essere assegnato univocamente ad uno specifico *master*, in fase di configurazione. Poichè ciascun nodo è identificato da un indirizzo univoco, la fase di configurazione permette ai nodi *master* anche di determinare le altre stazioni *master* attive e di stabilire l'anello "logico" di passaggio del token[5]. In effetti, un nodo *master* può comunicare con i propri *slave*, interrogandoli espressamente con un ciclo di "polling", solo quando detiene il token. Pertanto, il tempo di rotazione del token è un parametro molto importante per valutare l'efficienza della configurazione di una rete Profibus, che viene calcolato in base al numero di stazioni *master* che compongono l'anello virtuale. La configurazione più tipica di una rete Profibus-DP prevede in effetti un solo nodo *master*, tipicamente un PLC, per realizzare cicli di acquisizione dati tipici del controllo di macchina con moduli di I/O remoti. Nel caso che vi sia la necessità di collegare più dispositivi **master** alla rete, per semplificare l'architettura essi sono suddivisi in due classi: i *master* di classe 1 (tipicamente dispositivi di controllo), che possono comunicare direttamente con gli slave DP, ed i *master* DP di classe 2 (tipicamente dispositivi di programmazione e monitoraggio), che invece possono comunicare solo con i master di classe 1.

Come per il bus CAN, anche il frame Profibus può essere di quattro tipi (v. Figura B.3.19):

- di lunghezza fissa **senza** dati utente.
- di lunghezza fissa **con** dati utente.
- con campo dati utente di lunghezza variabile.
- il Token vero e proprio.

I frame iniziano pertanto con un delimitatore indicativo del tipo di messaggio, e terminano con un delimitatore di fine messaggio. I campi **Destination Address (DA)** e **Source Address (SA)** devono contenere indirizzi univoci nella rete, che vanno da 0 a 126 (max 127 nodi sulla rete). L'indirizzo 127 può essere usato nella fase di configurazione per la trasmissione di informazioni che sono indirizzate a tutti i nodi contemporanemante ("multicast"). Il campo per la correzione degli errori **Frame Check Sequence (FCS)** garantisce una **distanza di Hamming = 4**.

I protocolli Profibus-DP e PA permettono di realizzare i seguenti servizi:

[5]Cioè ogni nodo deve sapere da chi può ricevere il Token e a chi lo deve inviare, una volta terminato il tempo di permanenza del Token

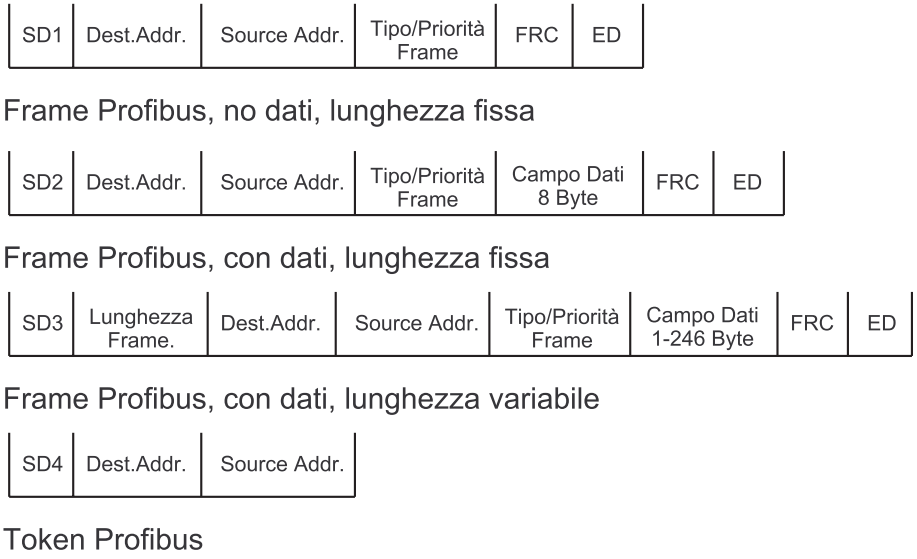


Figura B.3.19: Formati dei frame ammessi dal protocollo Profibus

- **SRD, Send and Request Data with reply**: consente ad un nodo di inviare dati richiedendone contemporaneamente al nodo destinazione. Quest'ultimo invia i dati contemporaneamente alla conferma dell'esito.
- **SDN, Send Data with No acknowledge**: per l'invio di dati senza conferma.

Profibus-FMS mette a disposizione anche i seguenti servizi aggiuntivi:

- **SDA, Send Data with Acknowledge**: un nodo invia dati e il ricevente invia un messaggio di sola conferma.
- **CSRD, Cyclic Send and Request Data with reply**: consente ad un nodo di avviare una comunicazione ciclica di dati con un altro nodo.

Come detto, un protocollo di livello 7 per Profibus è definito solo nel Profibus-FMS, che realizza una variante dello standard MMS. I servizi principali sono relativi:

- all'accesso a variabili di memoria del nodo
- al controllo remoto dell'esecuzione del software di un nodo (start, stop, reset, kill)
- al trasferimento di grosse aree di memoria (anche moduli di programma)
- alla gestione di eventi, tipicamente messaggi di allarme
- alla gestione dei canali di comunicazione virtuali.

Benchè per Profibus-DP e PA non esista un protocollo di livello 7, sono stati definiti dei **Profili Utente** che permettono di associare un determinato significato ai dati scambiati, in relazione al tipo di applicazione desiderata. I profili più importanti sono certamente:

- il **profilo per azionamenti a velocità variabile (PROFIDRIVE)**, che stabilisce come parametrizzare gli azionamenti e come trasmettere i riferimenti (di posizione o velocità). Con questo profilo è possibile realizzare sistemi di controllo assi con azionamenti di differenti costruttori.
- il **profilo per Human-Machine Interface (HMI)** stabilisce come collegare semplici apparecchiature di supervisione e pannelli operatore.

B.3.3.4 Confronto finale tra Profibus e CAN

Confrontando le reti Profibus con la rete CAN, si evidenzia come le caratteristiche dei due bus siano quasi completamente opposte: Master/Slave e accesso a Token contro Multi-Master e accesso CSMA/CA, indirizzamento Source/Destination contro Producer/Consumer, eccetera. Si è già detto che la rete Profibus è nata in un contesto differente, maggiormente orientato alle connessioni per moduli I/O remoti, per le quali i modelli Master/Slave sono più indicati. Infatti, la dimensione massima del campo dati di un pacchetto Profibus è molto maggiore (246 Bytes contro 8 Bytes) rispetto alla rete CAN. Tuttavia, la rete Profibus si rivela comunque poco efficiente qualora vi sia un notevole numero di nodi *master*, mentre la rete CAN si è rivelata più semplice da implementare (possibilità di realizzare controllori hardware del protocollo), ed ha avuto un maggiore successo nei sistemi "distribuiti", ma di piccole dimensioni, come quelli tipici dell'industria automobilistica (microcontrollori per sotto-sistemi separati: ECU per iniettori, ABS, ecc.), che trae notevoli benefici anche dalle maggiori capacità di rivelazione di errori del protocollo CAN (distanza di hamming = 6, contro il valore 4 per Profibus).