# Automazione Industriale – Controllori a Logica Programmabile (PLC)

**di Cesare Fantuzzi e Marcello Bonfè**

**Dipartimento di Ingegneria, Università di Ferrara**

Versione 1, 1 Dicembre 2000

# Capitolo 1

# Introduzione.

## 1.1 Architetture caratteristiche di sistemi di controllo a microprocessore.

Il controllo di processi industriali richiede che il sistema deputato al controllo abbia 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 (RS232, TCP/IP, ....) o con l'utente (tastiera, schermo).
- **Un sistema di acquisizione dati dal campo**, in grado di acquisire segnali analogici, digitali o logici a seconda della applicazione di controllo.
- **Un sistema di attuazione dei segnali di controllo**, comprendente i dispositivi 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 *custom***)
- oppure essere distribuiti su più schede collegate fra di loro da un *bus* di comunicazione[1] (**sistemi a *bus***).

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. 1.1 per un riassunto delle caratteristiche dei due sistemi).

I sistemi integrati (*custom*) sono di solito progettati ad hoc per controllare un semplice sottosistema, ad esempio un motore elettrico. Il progettista del sistema di controllo deve quindi realizzare 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 richiede particolari prestazioni, e quindi il controllo deve essere ottimizzato, o quando il complessivo sistema-controllo è venduto con elevata tiratura, e quindi il costo di progetto è ammortizzato da un minor costo per unità della soluzione *custom*.

Di contro, in genere, è difficile adattare i sistemi *custom* al controllo di altri dispositivi rispetto a quelli per cui sono stati progettati. Per queste loro caratteristiche i sistemi *custom* sono solitamente utilizzati al livello più basso, quello a stretto contatto con il processo, di un sistema di controllo gerarchico.

[1] Un *bus* di comunicazione è un insieme di linee elettriche su cui vengono trasmesse informazioni e comandi logici (dati, indirizzi, segnali di sincronizzazione)

I sistemi a *bus* non sono progettati per il controllo di un particolare sistema, ma hanno caratteristiche generali che consentono di adattarsi a diverse problematiche di controllo. In genere questi sistemi sono composti da un modulo principale a microprocessore e da schede di espansione che vengono scelte in base alle specifiche del sistema da controllare.

Le schede che compongono il sistema a *bus* vengono progettate e distribuite da produttori specializzati. Il progettista del sistema di controllo si limita a scegliere ed assemblare i componenti che al meglio soddisfano le specifiche richieste. Lo sforzo progettuale si conclude quindi nella scrittura del software di sistema che realizzerà le operazioni richieste dalle specifiche.

I sistemi a *bus* sono di solito usati per realizzare il livello più alto del sistema di controllo distribuito.

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

Tabella 1.1: Schema riassuntivo delle prestazioni dei sistemi *custom* e dei sistemi basati su un *bus*.

▽ *Esempio: Sistema di controllo basato su un PC.*

Il personal computer possiede un bus interno che può essere utilizzato per ospitare schede di acquisizione dati e per il controllo di attuatori. Il bus può essere di tipo ISA, EISA (bassa velocità) oppure PCI (alta velocità), ovviamente il secondo è adatto ai casi in cui si hanno tempistiche critiche relative al processo.

Data la diffusione dei sistemi a microprocessore di tipo Personal Computer, questo tipo di soluzione si avvantaggia di una grande quantità di software applicativo ed hardware disponibile in commercio a prezzi bassi.

Di contro questo sistema ha un certo numero di limitazioni dovute al fatto che il PC non è progettato per questo scopo. In particolare i dispositivi periferici (disco rigido, scheda video, schede di rete) possono occupare il sistema per tempi relativamente lunghi, rendendo impossibile l'esecuzione dell'algoritmo di controllo. Altre limitazioni sorgono da vincoli costruttivi: il numero di locazioni disponibili (*slot*) per schede di espansione è limitato, inoltre il PC prodotto per l'ufficio non è idoneo a lavorare in condizioni ambientali avverse (presenza di polvere, umidità, etc.), che però sono tipiche dell'ambiente industriale.

Per questo motivo sono stati sviluppati sistemi, i cosiddetti PC "industriali", che risultano compatibili in termini di hardware e software con i PC, ma che superano i vincoli sopra citati, ovviamente al prezzo di una lievitazione dei costi.

△

### 1.1.1 Sistemi di controllo digitale a logica programmabile (PLC)

Il Controllore Logico Programmabile (Programmable Logic Controller, PLC) è uno dei componenti fondamentali nell'Automazione Industriale. I motivi principali del suo successo sono da ricercarsi nella affidabilità e semplicità di programmazione, eseguita principalmente mediante semplici linguaggi di tipo grafico. Allo stato attuale il PLC è presente nella quasi totalità dei sistemi di automazione per impianti di media e grandi dimensioni.

Le origini della affermazione del PLC hanno anche radici di carattere storico. Infatti i primi controllori logici erano a struttura cablata, in cui dispositivi elettromeccanici quali relè e temporizzatori venivano utilizzati per implementare la logica di controllo di impianto. L'avvento della elettronica ha consentito di sostituire i dispositivi elettromeccanici, ingombranti e non riconfigurabili, con dispositivi elettronici in cui la logica di controllo veniva racchiusa nel software di programmazione del dispositivo, anzichè nella combinazione "hardware" di dispositivi elettromeccanici.

Vari autori concordano nel fissare la data di nascita del primo PLC nell'anno 1968, quando la General Motors diede le specifiche per una nuova generazione di controllori da destinare ai propri impianti, le più importanti riassumibili in:

- Caratteristiche di facile programmazione e riprogrammazione, eventualmente in sede di utilizzo.
- Facilità di manutenzione.
- Caratteristiche di espandibilità.
- Robustezza per poter funzionare con affidabilità in ambiente industriale, caratterizzato da interferenze elettromagnetiche, polvere, vibrazioni.

Il primo PLC basato su di un microprocessore fu introdotto negli anni '70 dalla Allen-Bradley (che utilizzava il 8080 dell'Intel), e, da allora, l'evoluzione dei PLC è proseguita di passo alla evoluzione dei microsistemi. Attualmente il PLC di fascia alta è un sistema multiprocessore con integrate funzioni di rete informatica evoluta, basato sulla stessa tecnologia di un Personal Computer tradizionale, però con caratteristiche idonee per l'ambiente industriale.

#### Architettura di un PLC

L'architettura di un PLC non si differenzia di molto da quella di un calcolatore "general purpose" ed è molto simile ai sistemi di controllo "a bus" analizzati in precedenza, infatti essa comprende (Figura 1.1):

- L'armadio, o 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, che esegue le elaborazioni necessarie al controllo di sistema.
- I moduli di ingresso/uscita, che permettono l'interfacciamento del PLC con i sensori e gli attuatori sul campo.
- Altri moduli aggiuntivi richiesti da particolari applicazioni, quali espansioni di memoria, moduli di conteggio, porte seriali sincrone ed asincrone, etc.

Il sistema si completa con un *terminale di programmazione*, che può essere un sistema dedicato oppure un PC, attraverso il quale il PLC viene programmato, ed un terminale di dialogo con l'operatore, attraverso il quale l'operatore inserisce i parametri di funzionamento del PLC.

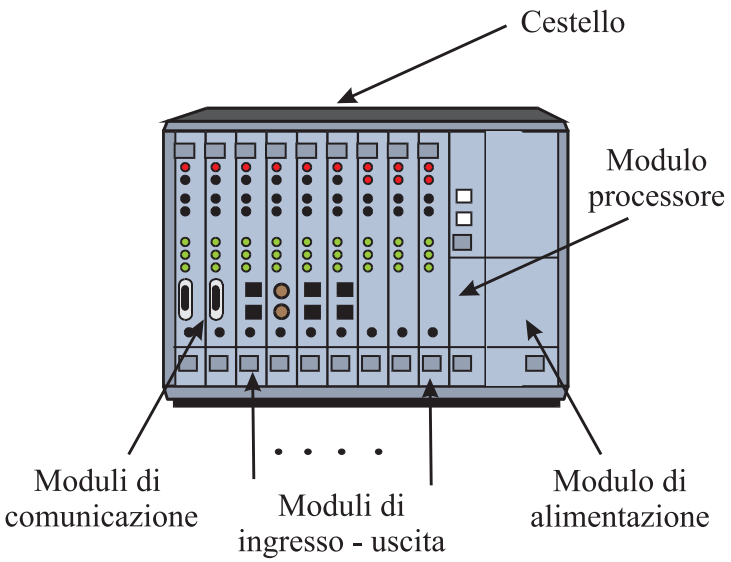



Figura 1.1: Architettura di base di un PLC.

### Il modulo processore

Il modulo processore rappresenta l'unità di governo del sistema PLC e racchiude una scheda con uno o più microprocessori, che eseguono il sistema operativo e i programmi utenti, la memoria di sistema e dispositivi per il controllo del bus di comunicazione con gli altri moduli.

Le caratteristiche che maggiormente denotano la natura specificatamente orientata all'automazione del PLC sono:

- La gestione automatica dei segnali di ingresso/uscita (I/O).
- La scansione ciclica del programma.

La gestione degli I/O è completamente automatica ed è affidata al Sistema Operativo (SO) della macchina. All'inizio del ciclo di esecuzione del programma il SO legge gli ingressi e li carica in una specifica sezione di memoria, che contiene così un'**immagine del processo** in quel momento. Questo consente di realizzare una acquisizione sincrona degli ingressi e di congelarla per tutto il ciclo di esecuzione del programma. Una volta completata la fase di acquisizione, può iniziare la fase di elaborazione del programma da parte della CPU. Durante questa fase vengono calcolati i valori delle uscite che sono memorizzati in una apposita sezione di memoria, e quindi non attuati immediatamente. Alla fine del ciclo di programma il SO interviene nuovamente per attuare in modo sincrono le uscite e per iniziare il ciclo di elaborazione successivo. Lo schema di un ciclo di elaborazione è mostrato in figura 1.2.

Oltre alla esecuzione ciclica del programma è previsto l'esecuzione immediata di moduli programma in risposta ad eventi mediante un meccanismo di interruzione gestito dal sistema operativo. Tali eventi possono essere esterni, come, ad esempio, il passaggio di stato di alcuni ingressi del PLC, ma più tipicamente sono eventi interni, come ad esempio, la caduta di alimentazione. In questo caso specifico, il sistema operativo reagisce salvando lo stato dell'elaborazione in una zona di memoria "tamponata" da una batteria.

Il **tempo di esecuzione** del programma fornisce una valutazione della velocità di elaborazione del PLC. Il tempo di esecuzione viene misurato in KWord di programma (1000 istruzioni) eseguite nell'unità di tempo, e quindi il tempo di ciclo del PLC dipende dalla lunghezza del programma. Come accennato in precedenza il controllo di processo richiede la determinazione temporale della esecuzione degli algoritmi di controllo. Per questo motivo il sistema operativo del PLC mette a disposizione un temporizzatore a cui agganciare il ciclo del programma, consentendo al programmatore la predeterminazione della durata del tempo di ciclo del PLC.

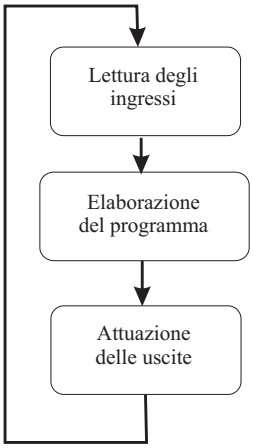



Figura 1.2: Schema del ciclo di elaborazione del PLC.

Il **sistema operativo** del PLC è costituito da un insieme di programmi di base per la gestione del programma utente, l'acquisizione degli ingressi e l'attuazione delle uscite, la diagnostica del sistema. Tra le altre operazioni è possibile ricordare la funzione di controllo del tempo di esecuzione del ciclo (Watchdog timer). Nel caso che esso superi un valore limite prefissato, significa che vi è stato un errore nel programma, perciò si attiva un opportuno modulo di risposta all'evento. Il sistema operativo comunica con il programma utente attraverso bit di memoria riservati.

#### I moduli di ingresso/uscita

I moduli di ingresso uscita costituiscono l'interfaccia del PLC con il processo da controllare. Questi moduli possono trattare segnali di tipo logico o analogico, anche se la maggior parte delle applicazioni nel campo delle macchine automatiche utilizza segnali di tipo logico.

Il motivo della presenza di moduli specializzati consiste nel fatto che i segnali generati dai trasduttori, anche quelli ad uscita logica, non possono venire direttamente interfacciati alla elettronica del modulo processore, a causa di differenze nel livello di segnale (tensione) o tipologia (segnale in tensione alternata, segnale in corrente[2]).

Ciascun ingresso ed uscita dei moduli di I/O (via di ingresso/uscita) è identificata dal sistema operativo del PLC in base alla posizione che il modulo occupa nel cestello. A livello operativo questo significa che non è possibile riconfigurare dinamicamente le vie del PLC ma queste sono fissate al momento del cablaggio dei collegamenti con i sensori.

I moduli assicurano le funzionalità di filtraggio (analogico e digitale) degli ingressi e la protezione contro sovratensioni ed inversione di polarità degli ingressi. Forniscono, inoltre, un indicatore di stato che mostra l'operatività della via di ingresso/uscita tramite un led luminoso ed un bit di stato di errore rilevato dal sistema operativo.

### 1.1.2 Sistemi di controllo distribuiti (DCS)

I sistemi di controllo distribuiti (Distibuited Control System, DCS), sono sistemi utilizzati per controllare processi di medie-grandi dimensioni che hanno la caratteristica di essere distribuiti geograficamente (es. un impianto di raffinazione petrolchimica).

Gli elementi che caratterizzano il DCS sono quindi la possibilità di distribuire in diversi punti moduli per l'acquisizione dati, l'elaborazione e il controllo, ed una rete di comunicazione il più possibile efficiente che metta in collegamento fra di loro i vari sottosistemi. Una caratteristica importante dei DCS consiste nella possibilità di modificare la topologia della rete, e quindi aggiungere e togliere moduli, con l'impianto in marcia.

[2]Il segnale trasmesso utilizzando una corrente è meno sensibile ai disturbi elettromagnetici presenti in ambiente industriale.

I moduli da cui il DCS è formato integrano essenzialmente:

- Una scheda di interfaccia della rete.
- Una scheda a microprocessore e memoria.
- Una o più schede specifiche per il compito che deve eseguire il modulo stesso.

La struttura hardware e software di base del DCS è studiata appositamente per poter governare un sistema di controllo con caratteristiche distribuite. In particolare si vuole astrarre completamente dal vincolo geografico a cui sono legati i moduli, secondo il principio in cui, ad esempio, il modulo che acquisisce i dati, quello che li elabora e quello che li attua possono essere distanti fra loro, senza che il programmatore del DCS debba curarsi di ciò.

L'astrazione della posizione geografica dei moduli è raggiunta considerando che ciascun modulo (acquisizione, controllo, attuazione) è proprietario (owner) di un certo "data-base" di informazioni, contenente la serie storica di alcune variabili di processo ("data-points") che il modulo controlla o supervisiona. Le informazioni contenute in un certo modulo sono visibili a livello globale e contrassegnate da un identificatore chiamato "tagname". Nel data base globale del DCS possono comparire "tagname" corrispondenti a diverse funzioni:

- identificatori per valori di ingresso (analogici, digitali, contatori, ....)
- Identificatori per moduli di elaborazione (Algoritmi PID, reti compensatrici, logiche combinatorie, ...),
- Identificatori per valori di attuazione in uscita (analogici, logici).

Quindi nel DCS ogni "tag" rappresenta una variabile di processo (es. FIC100), oppure un algoritmo o un parametro di uscita. Tutte le transizioni nel DCS avvengono sulla base di richieste del tipo "tag.parametro". Ad esempio quando un operatore richiede il movimento di una valvola, viene generato un messaggio di scrittura per il tag FIC100.OPEN. Il messaggio viene inviato, tramite la rete, al modulo che è proprietario (owner) del tag FIC100 che, dopo aver verificato le condizioni operative (es. se il controllo della valvola è in modalità automatica o manuale) eseguirà l'ordine impartito generando un messaggio di risposta.

## 1.2 Struttura software per i sistemi di controllo.

I programmi per il controllo automatico hanno una struttura caratteristica generale che risulta indipendente dalla particolare architettura scelta (si veda Fig. 1.3. Questa struttura ha come componenti fondamentali:

- La suddivisione del programma in **tre fasi**:
  - **Acquisizione degli ingressi**, in cui vengono acquisiti i dati forniti dai sensori sul campo.
  - **Elaborazione programma**, in cui i dati acquisiti dai sensori vengono elaborati mediante opportuni algoritmi di controllo.
  - **Aggiornamento delle uscite.**, in cui i dati elaborati vengono forniti al sistema di attuazione per esercitare una opportuna azione forzante sul sistema.
- Tali fasi sono **eseguite in sequenza e ciclicamente**. Le fasi di acquisizione, elaborazione aggiornamento uscite sono svolte ciclicamente secondo un periodo stabilito in sede di progetto. Tali attività sono da considerarsi *hard real-time* e quindi imprecisione sulla riesecuzioni periodica degli algoritmi possono portare a far fallire il sistema.
- In genere sono presenti attività di comunicazione con altri sistemi attraverso cui vengono scambiati comandi, parametri, stati del sistema controllato (ad esempio per memorizzare serie storiche per l'analisi della qualità della produzione) , etc. Tali attività sono da considerarsi *soft real-time* e quindi la precisione temporale della loro esecuzione non pregiudica la funzionalità del sistema.

I sistemi di controllo debbono interagire con processi reali che hanno un loro stato interno, che è rappresentabile attravero il valore di alcune variabili caratteristiche dette *variabili di stato*. Quindi i programmi per il controllo debbono necessariamente prevedere la possibilità di memorizzare variabili interne di stato a cui corrisponde uno stato del processo sotto controllo.

In particolare, alla accensione del sistema di controllo lo stato del processo da controllare non è, in generale, noto a pariori, quindi occorre prevedere un programma di servizio che porta il sistema in uno stato noto attraverso una serie di operazioni da cui partirà l'elaborazione normale del controllore. L'operazione di avvio (*start-up*) può essere di due tipi:

- **Avviamento a freddo**, in cui il processo deve essere completamente inizializzato e portato in una fase propizia per l'inizio della produzione vera e propria.
- **Avviamento a caldo**, in cui il processo riparte dopo una breve interruzione (ad esempio causata da una breve interruzione dell'energia elettrica). In questo caso è posssibile riprendere a produrre dopo una procedura di inizializzazione ridotta.

▽ *Esempio: Avviamento a caldo ed avviamento a freddo.*

Nei processi in cui occorre portare il prodotto ad una certa temperatura per la lavorazione (ad esempio ceramiche e piastrelle), la procedura di avviamento a freddo consiste nel portare a temperatura il forno.

Se una breve interrruzione dell'energia elettrica comportasse un fermo della produzione senza diminuire significativamente la temperatura del forno, allora sarà conveniente effettuare un avviamento a caldo, in cui la procedura di riscaldamento del forno non viene richiamata.
△

Allo spegnimento del sistema occorre prevedere una procedura per la fermata ordinata del sistema. Tipicamente il processo sul prodotto è di tipo "a catena di montaggio", in cui il prodotto è lavorato da diverse sottoparti della macchina in sequenza. Ovviamente non sarà conveniente fermare la macchina in uno stato in cui le sottoparti sono cariche di prodotto, ma occorrerà svuotare in sequenza le sottoparti in modo da fermare la macchina in uno stato di riposo.

Questi concetti sono basilari nella stesura di un corretto programma per un sistema di controllo.

## 1.3 Sistemi di comunicazione per architetture di controllo distribuite

I sistemi di controllo distribuiti necessitano di un sistema di comunicazione ad elevata banda passante e molto affidabili. Le reti di comunicazioni si possono distinguere in tre categorie (si veda anche Figura 1.4):

- **Le reti informatiche**, che collegano i sistemi di supervisione di alto livello con altri sistemi informativi di azienda. Su queste reti vengono scambiate informazioni relative alla produzione di stabilimento utili alla gestione del prodotto finito, alle scorte di magazzino e alla contabilità generale dell'azienda.

  La trasmissione di dati in questo tipo di reti non deve soddisfare specifiche di tempo reale, e quindi possono essere utilizzati protocolli di comunicazione standard in modo da facilitare l'integrazione di differenti sistemi. Due esempi di protocolli utilizzati sono il TCP/IP, legato inizialmente all'ambiente UNIX e ora ampiamente diffusa quale protocollo standard per la rete globale "internet", ed il protocollo DECNET relativo alle reti di calcolatori Digital Equipment.

- **Le reti per il controllo**, utilizzate per assicurare la comunicazione tra i dispositivi dedicati al controllo e alla supervisione degli impianti. In questo caso le informazioni debbono essere scambiate in modo da:

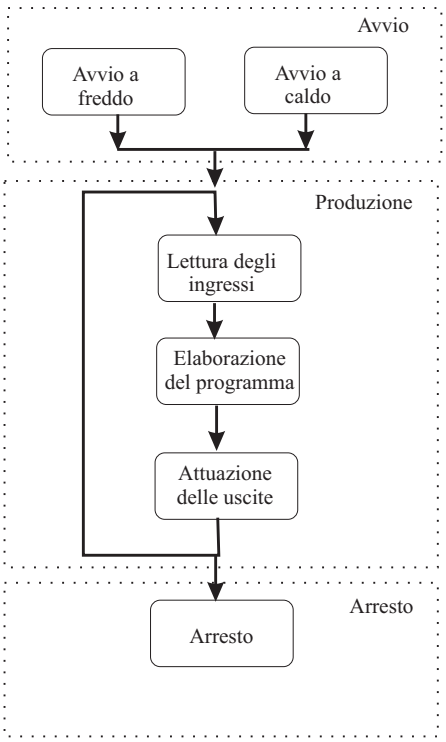



Figura 1.3: Struttura tipica di un programma per il controllo.

- Garantirne l'assenza di errori.
- Rispettare determinati vincoli temporali sulla durata della comunicazioni (Real Time Network).

infatti queste reti sono utilizzate per scambiare informazioni tra i sistemi di controllo (ad es. PLC) in modo da sincronizzare le operazioni tra sottoparti di un processo produttivo, e quindi per garantire la sincronizzazione degli eventi occorre che la trasmissione delle informazioni sia temporalmente deterministica.

Le realizzazioni di reti per il controllo sono in genere di tipo "proprietario", cioè sviluppate dallo stesso produttore dei PLC di controllo, e possono integrare tra di loro i PLC di quel costruttore e Personal Computer che siano dotati di schede e programmi appositi per l'interfacciamento. Nelle reti di controllo manca un'opera di standardizzazione, per cui risulta molto difficile integrare su di una stessa rete sistemi di controllo di costruttori differenti.

- **Le reti di campo (fieldbus)**, utilizzati per scambiare informazioni tra i sistemi di controllo (ad es. PLC) e i sensori ed attuatori dotati di una interfaccia digitale.

  Questo tipo di configurazione dell'architettura del sistema di controllo è di recente introduzione. La configurazione tipica di un sistema di controllo allo stato attuale è quella in cui i sensori e gli attuatori sono collegati al controllore direttamente, attraverso i moduli di ingresso/uscita del controllore stesso.

  Nella rete di campo i sensori e gli attuatori vengono forniti di moduli di comunicazione basati su di un microprocessore (per questo si usa dire che sono sensori/attuatori "intelligenti") e collegati al sistema di controllo attraverso una rete digitale (fieldbus).

  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 e manutenzione dei cavi.
– La possibilità di trasmettere informazioni di alto livello, come ad esempio funzioni di autodiagnosi, funzioni di interrupt.
– La possibilità di configurare i sensori e gli attuatori direttamente sul campo.
– Una maggiore robustezza delle trasmissioni, in quanto la trasmissione digitale è intrinsecamente meno sensibile ai disturbi rispetto a quella analogica.

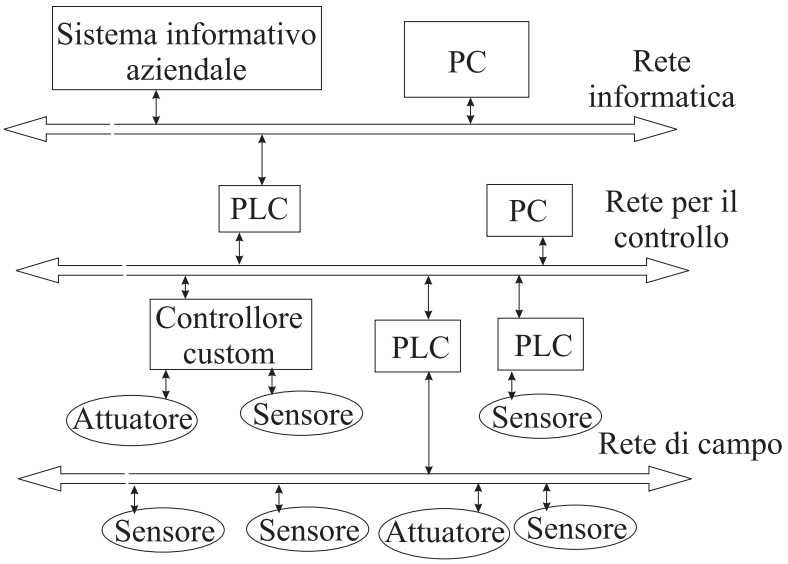



Figura 1.4: Reti per l'automazione

Il vero ostacolo alla diffusione delle reti di campo consiste nella scarsità di dispositivi (sensori ed attuatori) presenti sul mercato con interfaccia compatibile alla rete di campo.

Questo ritardo non è dovuto a difficoltà tecnologiche ma a motivi commerciali. Infatti al momento non vi è uno standard riconosciuto relativo alla rete di campo, e quindi, in questo, la situazione è molto simile alla rete di controllo. Tuttavia, mentre nel secondo caso i dispositivi sono prodotti da una stessa casa costruttrice, e quindi l'assenza di uno standard è meno sentita, nel primo caso invece gli elementi del sottosistema (sistemi di controllo, sensori ed attuatori) sono prodotti generalmente da costruttori diversi e quindi la mancanza di uno standard incide profondamente sulla possibilità di assemblare i componenti in un sistema integrato.

## 1.4 Sistemi di supervisione ed acquisizione dati (SCADA)

La gestione di impianti di medie e grandi dimensioni richiede l'acquisizione ed elaborazione di un elevato numero di dati.

Si pensi ad esempio la necessità di gestire le scorte di magazzino e le spedizioni ai clienti del prodotto finito. Per poter pianificare queste attività in modo sincrono rispetto alla produzione, e quindi per evitare fermi macchina dovuti alla mancanza di materie prime, oppure ritardare inutilmente la spedizione del prodotto, occorre che i manager dell'azienda abbiano un rapporto dettagliato e costantemente aggiornato delle attività produttive della fabbrica.

Altre esigenze che richiedono un costante controllo sui dati relativi alla produzione, consiste nella gestione degli allarmi di macchina e dei parametri di lavorazione. Infatti la produzione di un impianto è basata su molteplici sottosistemi, per cui l'eventualità che se ne guasti uno è tutt'altro che remota. L'operatore di processo, che in genere non è un esperto di sistemi di automazione,

deve poter agire in modo semplice su questa condizione di emergenza ed eventualmente recuperare il normale funzionamento dell'impianto.

La richiesta comune sulle specifiche dei sistemi di automazione più moderni consiste nella possibilità di variare alcuni parametri di lavorazione in maniera semplice, agendo solamente sul software di sistema. Ad esempio una macchina che impacchetta sia il latte che lo yogurt sarà preferibile ad una dedicata ad uno solo dei due prodotti. Tali parametri debbono essere modificabili dall'operatore di fabbrica in modo semplice e senza possibilità di compromettere la produzione nel caso di immissione di dati sbagliati.

In sostanza occorre un sistema semplice, utilizzabile da non esperti di progettazione di sistemi di controllo, in grado di fornire informazioni chiare e dettagliate e consentendo modifiche dei parametri di lavorazione durante il normale funzionamento della macchina. Questi sistemi sono chiamati **sistemi di supervisione ed acquisizione dati**, o utilizzando termini anglosassoni, **Supervisory Control And Data Acquisition (SCADA)**. Le caratteristiche di un sistema SCADA possono essere così riassunte:

- La possibilità di acquisire dati dal campo, eseguendo elaborazioni quali il calcolo di medie, formattazione di rapporti periodici, memorizzazioni dei dati in serie storiche, etc.
- La funzione di modifica dei parametri di lavorazione, ad esempio il set point di temperatura in una cella frigorifera.
- La possibilità di gestire allarmi, attivando procedure di correzione dell'evento che ha prodotto l'allarme, oppure provvedendo ad eliminare la funzionalità che ha prodotto l'errore (produzione incondizionata).
- Presentare una interfaccia operatore che mostri in maniera chiara lo stato di funzionamento dell'intero sistema, che consenta l'introduzione di parametri di lavorazione e la gestione di situazione di allarme. L'interfaccia operatore è in genere costituita da pagine grafiche che rappresentano pannelli di comando e quadri sinottici dell'intero impianto o di parti di esso.

  I pannelli di controllo permettono all'operatore di inviare comandi all'impianto attraverso simboli grafici che rappresentano pulsanti, manopole comandi a slitta, etc. Nei quadri sinottici si evidenzia la presenza di elementi statici, i cui attributi (aspetto, colore, dimensione, etc.) non variano, e di elementi dinamici, per cui tali attributi variano a seconda del valore di certe variabili di processo (ad esempio una pompa può divenire rossa o meno per segnalare il suo stato di funzionamento).
- Il progetto del sistema SCADA deve tenere conto del fatto che l'operatore dell'impianto non è in genere un esperto di controlli automatici, e non ha nessuna conoscenza della reale implementazione del sistema di controllo, che in genere è progettato e implementato da aziende esterne alla fabbrica produttrice che utilizza l'automatismo.

  Per questo motivo l'interfaccia deve essere semplice, chiara e l'introduzione di parametri deve essere a prova di errori, nel senso che un uso non corretto dell'intefaccia non deve produrre un blocco della produzione. Quando il processo richieda di inserire dati critici in relazione al funzionamento dell'automatismo, occorrerà che questi siano protetti da password ed autorizzazioni all'accesso.

# Capitolo 2

# Lo Standard IEC 61131-3: elementi comuni

## 2.1 Introduzione

La progettazione e lo sviluppo del software di controllo per PLC (Programmable Logic Controller) è un compito che presenta alcune problematiche dovute principalmente alla sua dipendenza dall'hardware del controllore da programmare. Il mercato dei controllori programmabili per l'industria offre una grande varietà di offerte, da parte di una moltitudine di produttori differenti. Ognuno di questi produttori mette a disposizione dei propri clienti un ambiente di sviluppo proprietario, talvolta diversificato persino per serie diverse dello stesso costruttore. Sebbene questi tools siano simili tra loro, soprattutto per quanto riguarda gli strumenti essenziali (editor di linguaggio a contatti, generatore di liste incrociate dei riferimenti in memoria ecc.), le difformità tra le architetture hardware dei controllori e le implementazioni specifiche di funzionalità analoghe fanno sì che il set di istruzioni a disposizione del programmatore possa variare notevolmente, costringendo il progettista ad un attenta lettura dei manuali prima di poter sviluppare un'applicazione su un controllore non conosciuto.

Inoltre, la necessità di sviluppare linguaggi diversi dal semplice linguaggio a contatti (linguaggi grafici o testuali di alto livello) per permettere al programmatore di risolvere compiti più complessi di semplici operazioni booleane, sfruttando nel contempo l'incremento delle prestazioni computazionali ottenibili con il miglioramento tecnologico, ha portato ad una ulteriore diversificazione tra i produttori e tra gli ambienti di sviluppo. Allo stato attuale, si riscontra l'assenza di una standardizzazione completa nei linguaggi e nelle strutture di programmazione per tali controllori. Diretta conseguenza di queste difformità nel panorama della produzione di PLC sono:

- l'impossibilità di portare codice pre–esistente tra macchine differenti;
- l'assenza di strumenti di sviluppo comuni.

È evidente che poter superare queste limitazioni ridurrebbe di molto la fase di progetto ed implementazione del software di controllo di una macchina, soprattutto in una ditta che costruisce macchine "su commessa", lavorando sulle specifiche fornite dal cliente il quale, tipicamente, impone anche la scelta dei dispositivi fisici per il controllo. Per favorire un punto di incontro tra i progettisti del controllo e i produttori di controllori industriali, negli anni recenti è stato introdotto dall'organismo internazionale IEC (Internation Electrotechnical Commission) uno standard che si propone di definire gli aspetti descrittivi e di programmazione dei dispositivi di controllo per l'Automazione Industriale. Tale documento, nominato IEC 61131, ed in particolare la sua parte Terza "Programming Languages" è oggetto dell'analisi dei prossimi tre capitoli.

## 2.2 Le origini dello Standard IEC 61131-3

La crescente complessità dei sistemi e controllori programmabili, la pluralità di configurazioni hardware e software dei PLC, hanno reso sempre più forte l'esigenza, sentita sia dai costruttori sia dagli utenti, di una standardizzazione. Grazie alla spinta di entrambi, sin dal 1988, nell'ambito della Comissione Elettrotecnica Internazionale (IEC), il comitato TC65 (Comitato Tecnico per la Misura e Controllo degli Impianti Industriali) ha dato il via a diverse Task Force (TF) con il compito di elaborare un nuovo standard per uniformare le caratteristiche dei sistemi di controllo industriale. In particolare, nel sottocomitato TC65B vi è il Working Group 7 (WG7) preposto ai controllori programmabili. L'opera di quest'ultimo si concretizza, a partire dal 1992, nella norma 1131; la parte terza di tale norma viene emessa nel 1993.

Nel corso del 1999 ne è stata iniziata una revisione, e nel contempo due sono stati i Technical Reports (TR) emessi: il TR 2 tratta le correzioni/estensioni, mentre il TR 3 (del quale è disponibile un Final Draft) riguarda le linee guida all'implementazione della parte terza.

Lo standard IEC 1131-3 fornisce una definizione dei linguaggi di programmazione per i controllori a logica programmabile, allo scopo di far evolvere verso una normalizzazione di tali linguaggi in modo che il programmatore possa astrarre il più possibile dall'architettura del particolare controllore che verrà utilizzato. Inoltre, lo standard definisce una serie di caratteristiche comuni a tutti i linguaggi, relative in particolar modo alla sintassi di dichiarazione di variabili simboliche e di moduli software, che consentono di orientare la programmazione ad un livello di astrazione più elevato. E' evidente che il raggiungimento di tale obiettivi porta alla risoluzione dei problemi elencati precedentemente, ed in particolare consente una programmazione di tipo strutturato e modulare, conducendo ad un maggiore facilità di riuso del software.

Si ritiene utile notare che lo standard IEC 1131-3 trova la sua applicazione naturale nel panorama dei sistemi di controllo basati su PLC, ma le caratteristiche generali della sua definizione ne suggeriscono applicazioni molto interessanti anche nelle soluzioni basate su PC[1] o su altri tipi di controllori elettronici. Dalle definizioni fornite nel seguito, risulta evidente che i dispositivi cui si fa riferimento sono identificabili nei PLC più per le consuetudini radicate che per associazioni esplicite.

#### Definizioni di Controllore Programmabile

Secondo lo standard IEC 1131 il **controllore a logica programmabile** è:

> *Un sistema elettronico a funzionamento digitale, destinato all'uso in ambito industriale, che utilizza una memoria programmabile per l'archiviazione interna di istruzioni orientate all'utilizzatore per l'implementazione di funzioni specifiche, come quelle logiche, di sequenzializzazione, di temporizzazione, di conteggio e di calcolo aritmetico, e per controllare, mediante ingressi ed uscite sia digitali che analogiche, vari tipi di macchine e processi.*

mentre il **sistema** controllore a logica programmabile è:

> *la configurazione realizzata dall'utilizzatore, formata da un controllore a logica programmabile e dalle periferiche associate, necessaria al sistema automatizzato previsto.*

Nell'uso comune si indica con il termine PLC sia il PLC vero e proprio, cioè la scheda processore, sia l'intero sistema completo delle sue schede di interfaccia. Nel seguito si indicherà con il termine PLC:

> *L'insieme dei dispositivi hardware che compongono un sistema di automazione basato su un controllore a logica programmabile.*

[1] si veda la crescente popolarità di ambienti di sviluppo come IsaGraf della CJ International (www.isagraf.com).

## 2.3 Il modello software

Nel corso dell'elaborazione dello standard 1131-3, il WG65 della IEC, ha dovuto analizzare tutto il contesto della programmazione dei PLC, considerando anche le interazioni tra il programma e l'ambiente operativo del sistema di controllo:

- Le interfaccie di Input/Output;
- Le interfaccie di Comunicazione con altri PLC, pannelli operatore, dispositivi di programmazione;
- Le interfaccie di Sistema, tra il programma del PLC e l'hardware del PLC stesso (inizializzazioni, messa in esecuzione ecc.).

Per poter analizzare correttamente il problema della programmazione del controllo di un sistema complesso[2], occorre considerare l'insieme di tutti i dispositivi utilizzati.

In questo senso la IEC ha elaborato un *modello* del sistema di controllo, definendone i suoi componenti *software*.

Il modello software proposto dallo standard IEC 61131-3 è di tipo stratificato e gerarchico, e si basa su di un insieme ben preciso di elementi, ognuno dei quali si trova ad un determinato livello della gerarchia e "racchiude" gli elementi del livello sottostante. Tale gerarchia, può essere schematicamente descritta dalla figura 2.1.

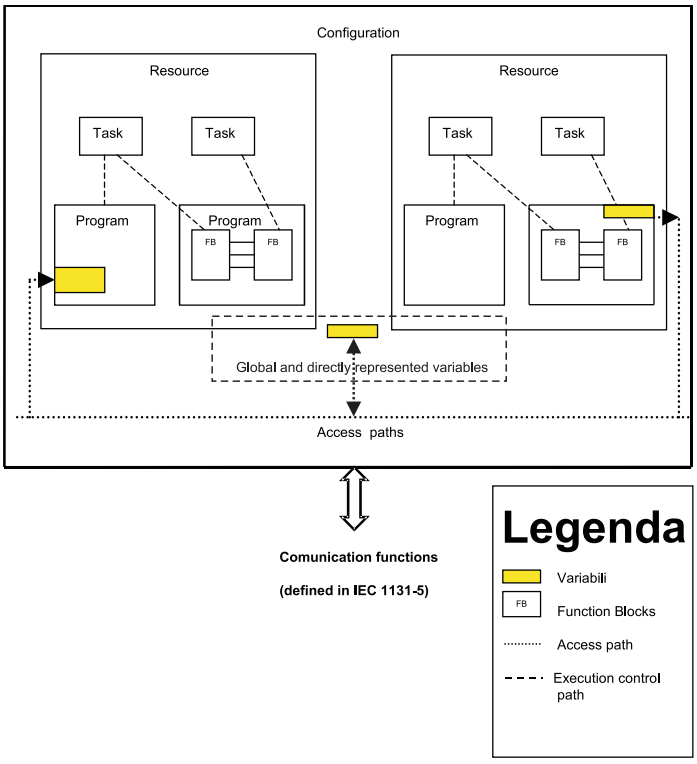



Figura 2.1: Schematizzazione del modello software IEC 1131–3

Nel presentare le componenti del modello software si utilizzeranno i termini originali in inglese, poichè il loro impiego nell'ambito dello standard può differire dal significato ad essi comunemente attribuito.

[2]spesso basato su numerosi PLC, diversi tipi di Bus di Campo, attuatori "intelligenti" ecc.

### 2.3.1 Gli oggetti del modello software

I componenti del modello software IEC 1131-3 sono, come detto, distribuiti secondo una struttura gerarchica che può essere descritta come un sistema ad oggetti, nel quale ogni elemento è dotato di caratteristiche ed attributi propri, è connesso ad altri oggetti secondo opportune relazioni di aggregazione, appartenenza o utilizzo. Nel seguito sono fornite le definizioni di tali oggetti, mettendo in evidenza, in modo testuale, le relazioni espresse graficamente nella figura 2.1.

#### Configuration

È l'elemento più esterno della struttura, perciò contiene gli altri elementi, e corrisponde all'intero sistema controllore programmabile[3], ovvero, generalmente ad un PLC. In un sistema di controllo è possibile avere più configuration, le quali possono comunicare conformemente alle modalità previste e definite nella parte Quinta dello standard IEC 61131 ("Communications"). Ad esempio è possibile incontrare architetture di sistemi di controllo nelle quali più PLC sono connessi in rete, tramite Bus di Campo (Fieldbus).

#### Resource

Sono contenute all'interno di una configuration: esse rappresentano il supporto di esecuzione dei programmi, o in altri termini, l'interfaccia di una "macchina virtuale" in grado di eseguire un programma IEC, il quale per poter funzionare deve necessariamente essere caricato in una resource. Quest'ultima, usualmente, è individuabile internamente al PLC, ma potrebbe essere anche simulata con un PC, oppure, ancora, individuabile all'interno di un PC.

Lo standard fornisce una descrizione concisa di Resource: "Un elemento che corrisponde ad una unità funzionale di elaborazione dei segnali, connessa a dispositivi di interfaccia uomo-macchina e con funzioni di interazione con sensori ed attuatori". Quindi, una delle principali funzioni di questa componente del modello software è di fornire un'interfaccia fra il programma e gli I/O fisici del PLC, oltre che con l'uomo (tramite i dispositivi di programmazione[4]).

Le Resource sono, in generale, componenti del modello software tra loro autonome, in quanto possono essere reinizializzate indipendente e separato, e sono, solitamente, poste in corrispondenza biunivoca con un diverso processore (CPU). Ciò nonostante, è altresì possibile l'accesso diretto di queste componenti alle *Global Variables* ed alle *Directly Represented Variables* (v. seguito) definite a livello di Configuration, le quali, quindi, costituiscono un meccanismo di comunicazione fra le Resource.

E' interessante osservare che lo Standard, prevedendo a livello più esterno una Configuration che può contenere più resource, le quali a propria volta possono supportare più Program, conferisce al PLC la possibilità di caricare ed eseguire un certo numero di programmi **totalmente indipendenti**, se esistono "risorse" sufficienti: in futuro, questa situazione potrebbe, con il progredire della tecnologia ed il ridursi dei costi, divenire una consuetudine.

#### Program

Lo standard IEC 1131-3 definisce quest'unità nel seguente modo: "L'insieme logico di tutti gli elementi di programmazione ed i costrutti necessari per la funzionalità di elaborazione di segnali richiesta ad un sistema controllore programmabile." I Program si trovano necessariamente all'interno di una Resource. Essi sono i contenitori dei costrutti realmente eseguibili, programmati nei linguaggi previsti dallo standard. L'esecuzione di un Program avviene sotto il controllo di uno o più Task (v. seguito); nella situazione più semplice nessun Task controlla il Program, ed in tal caso quest'ultimo è eseguito

[3] secondo la definizione generica IEC precedentemente esposta

[4] storicamente i dispositivi di programmazione erano tastierini alfanumerici connessi tramite un protocollo proprietario alla CPU del PLC. Ora si tratta quasi sempre di un Personal Computer connesso tramite cavi seriali alla CPU del PLC.

seguendo le specifiche di default del costruttore[5]. Nelle realizzazioni più complesse, possiamo, al contrario, avere diversi Tasks che controllano l'esecuzione dei diversi Program.

### Task

È il componente del modello software preposto al controllo dell'esecuzione di Programs (o di loro parti[6]): per ciascuno di essi può essere specificata un'attivazione periodica o al verificarsi di un determinato evento. Quindi, una volta attivati, i Task mandano in esecuzione i moduli dell'applicazione ad essi associati, con le tempistiche o le attivazioni "ad interrupt" imposte.

### Local and Global Variables

Gli identificativi delle variabili sono dei nomi simbolici cui il programmatore attribuisce un determinato significato, dipendente dal suo tipo di dato e dalla zona di memoria in cui si richiede che sia allocata. Possono essere definite come locali, quindi accessibili unicamente dalle unità nelle quali sono dichiarate (Programs e moduli di programma, Resources, Configurations). Tuttavia, le variabili possono essere anche definite globali: ciò implica la loro accessibilità da parte dell'elemento in cui sono dichiarate ed, inoltre, di tutti quelli in esso contenuti, secondo la struttura gerarchica definita nel modello software IEC precedentemente introdotto. In quest'ultimo caso, è evidente che le variabili (globali) forniscono un supporto al trasferimento di dati ed informazioni tra program o tra FB residenti in differenti program.

### Directly represented variable

Oltre alle variabili riferite attraverso nomi simbolici, è possibile indirizzare locazioni di memoria (di sistema, v. sezione 2.3.4) del PLC utilizzando variabili di questo tipo. Allo scopo di mantenere un soddisfacente grado di riutilizzabilità, lo Standard suggerisce un uso di queste variabili limitato al livello di Program. Scendendo di livello, avere Directly Represented Variables comprometterebbe la portabilità del codice su differenti progetti.

### Function Blocks

È il concetto introdotto dallo standard IEC 1131-3 in grado di supportare una progettazione gerarchica del software di controllo. Essi sono moduli di programma riutilizzabili, grazie ai quali è possibile ottenere programmi strutturati. Due sono le principali componenti di un Function Block:

- DATI: l'insieme delle variabili utilizzate solamente dal codice contenuto nel Function Block, costituito dai parametri in ingresso ed uscita, e dalle variabili interne (merker, variabili temporali ecc.);
- ALGORITMO: un insieme di istruzioni che costituiscono il corpo del modulo di programma, mandate in esecuzione ogni volta che il relativo Function Block è richiamato.

L'algoritmo processa il corrente valore degli ingressi e delle variabili interne, producendo così un nuovo set di valori per i parametri d'uscita e le variabili interne. Queste ultime possono essere locali o globali: nel primo caso esse sono dichiarate internamente al Function Block, nel secondo sono dichiarate a livello di Program, Resource o Configuration. Quindi, i Function Block possono essere visti come "dispositivi software" riutilizzabili, dotati di uno stato interno, tra loro interconnessi tramite i "morsetti" d'ingresso e di uscita.

L'IEC 1131-3 definisce inoltre un insieme di Function Block standard, tra i quali i temporizzatori, i contatori, etc.

[5] solitamente il programma viene eseguito nel minor tempo di ciclo possibile. Talvolta questa situazione porta ad avere tempi di scansione degli I/O molto incostanti, il che può avere effetti negativi sul funzionamento dell'automatismo da controllare. Pertanto imporre un Task primario ciclico e con tempo prefissato risulta sempre una buona consuetudine.

[6] infatti secondo lo Standard è possibile assegnare a Tasks differenti anche differenti moduli dello stesso Program

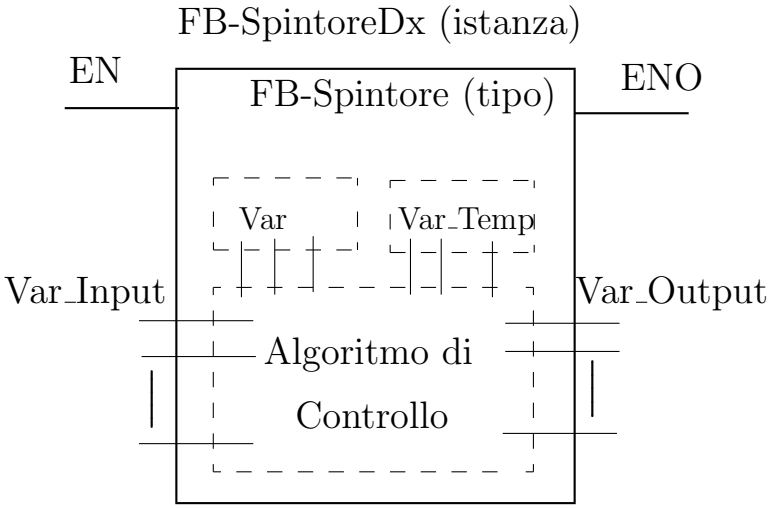



Figura 2.2: Rappresentazione grafica di un FB.

**Functions**

Le Functions sono anch'esse elementi software riutilizzabili, che ricevono in ingresso un set di valori, e producono in uscita un solo valore. Questa è una prima caratteristica che le differenzia dai Function Blocks. Inoltre, mentre questi ultimi possono contenere le dichiarazioni di variabili statiche, cioè il cui valore viene mantenuto tra due esecuzioni successive, le variabili interne di una Function possono essere solo temporanee. In sostanza, mentre i Function Blocks sono dotati di un proprio stato interno, le Function mancano di tale proprietà, per cui, in corrispondenza della stessa configurazione d'ingresso esse forniscono sempre lo stesso risultato.

Anche per esse l'IEC 1131-3 specifica un insieme di funzioni standard, principalmente per effettuare conversioni tra differenti tipi di dato e per effettuare elaborazioni matematiche.

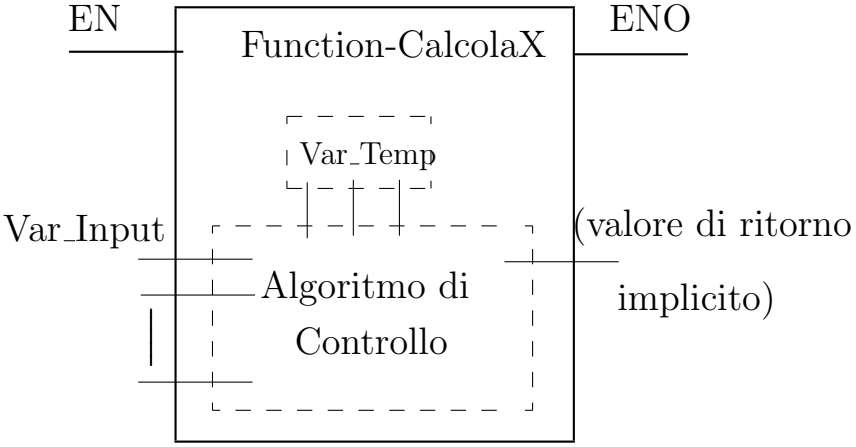



Figura 2.3: Rappresentazione grafica di una Function

**Access path**

È l'ultimo degli elementi del modello software introdotto dallo standard. Poichè un sistema di

controllo può comporsi di diverse Configuration, le quali devono scambiarsi dati ed informazioni, è necessario supportare tale trasferimento con opportuni protocolli.

Ad esempio, esse possono comunicare attraverso reti basate su Ethernet, Fieldbus o "backplane bus" di tipo proprietario. Lo scambio di informazioni a questo livello esula però dall'IEC 61131-3[7]. E' attraverso la specifica di un set di queste variabili "speciali", dichiarate utilizzando il costrutto `VAR_ACCESS`, che vengono definiti gli access path: in tal modo il progettista definisce l'interfaccia, costituita dal suddetto set di variabili, che una configuration presenta ad altre configuration remote, le quali possono accedere a tali variabili.

### 2.3.2 Architettura del software di controllo

Il modello software precedentemente descritto permette una descrizione completa del sistema di controllo per un automatismo. Alcuni dei concetti descritti (Programs, Function Blocks, Functions) sono riscontrabili nella pratica attuale in alcuni ambienti di sviluppo, che supportano la programmazione strutturata e modulare. Altri concetti (Configuration, Resource, Tasks e Access Paths) richiedono, come detto un'analisi più approfondita dell'architettura globale di un sistema di controllo e delle sue possibili estensioni grazie agli sviluppi tecnologici.

Le definizioni di *Configuration, Resource* e *Program*, danno modo di costruire lo scheletro dell'architettura del sistema di controllo, che costituisce la piattaforma sulla quale eseguire il programma vero e proprio. La Configuration, contiene la descrizione d'insieme del sistema di controllo, con le diverse CPU utilizzate, rappresentate dalle Resource, e l'insieme dei programmi eseguiti. Tali programmi possono anche essere più di uno sullo stesso processore, se il dispositivo supporta il **multitasking**[8]. In questo caso la norma definisce come questi programmi possano essere associati a diversi Task in esecuzione contemporanea ed anche la sintassi per la dichiarazione delle modalità di attivazione dei task[9]

Nel dettaglio, possibili esempi di dichiarazione delle componenti la configurazione di un sistema controllore possono essere i seguenti:

- Una Resource deve essere associata ad una unità di elaborazione (CPU) e contenere le dichiarazioni di quali task essa elabora e quali programmi associati a queste task, pertanto un'esempio di dichiarazione aderente alla sintassi IEC 61131-3 potrebbe essere:

```
RESOURCE Res1 ON PROC_8044
        TASK Mast_task
             (INTERVAL:= T#50ms, PRIORITY:=0);
        TASK Slow_task
             (INTERVAL:= T#200ms, PRIORITY:= 1);
        PROGRAM turbine1 WITH Mast_task: turbine(
                        (* eventuali parametri di I/O...  *)
                        );
        PROGRAM turbine2 WITH Slow_task: turbine(
                        (* eventuali parametri di I/O... *)
                        );
END_RESOURCE
```

  Si noti che i programmi `turbine1` e `turbine2` sono due istanze dello stesso programma di tipo `turbine`, ma operano con tempi di ciclo differenti.

- La dichiarazione dei Task precedenti contiene l'intervallo di tempo di esecuzione e la priorità. Una dichiarazione alternativa può essere:

[7] la trattazione delle comunicazioni tra dispositivi di controllo fa parte della parte 5 della norma (IEC 61131-5), nella quale sono definiti alcuni Function Blocks standardizzati per lo scambio dati tra sistemi remoti.

[8] per una panoramica dei concetti di schedulazione di processi in un'ambiente aderente alla Norma IEC 61131-3 si veda il capitolo 3 di [3]

[9] la politica di scheduling è di libera implementazione da parte del costruttore del controllore.

```
TASK Event_task
     (SINGLE:= Event_alarm, PRIORITY:= 0);
```

In questo caso il Task si attiva quando il valore booleano del suo parametro `SINGLE` (associato all'evento di allarme) diventa vero.

- La configuration quindi dovrà contenere le dichiarazioni delle resource ed eventualmente delle variabili di accesso remoto da e per altre configuration:

```
CONFIGURATION unit_1
      RESOURCE res1 ON PROC_8044
             TASK... (* Vedi sopra *)

      RESOURCE res2 ON PROC_8045
             ...     (* Vedi sopra *)
      VAR_ACCESS
           UNIT_1_START: res1.turbine1.start: BOOL READ_WRITE;
           UNIT_1_ALARM: res1.turbine2.alarm: BOOL READ_ONLY;
           ..
      END_VAR
```

Nelle applicazioni relative a macchine automatiche di medie complessità, la configurazione del controllore maggiormente riscontrabile nella pratica è quella basata su un PLC di medie dimensione con un unica CPU e un unico programma (vedi figura 2.4), e raramente è prevista la possibilità di definire diversi task. In questo caso, ovviamente, il modello software si semplifica, e i concetti descritti nelle dichiarazioni esposte precedentemente vengono a mancare.

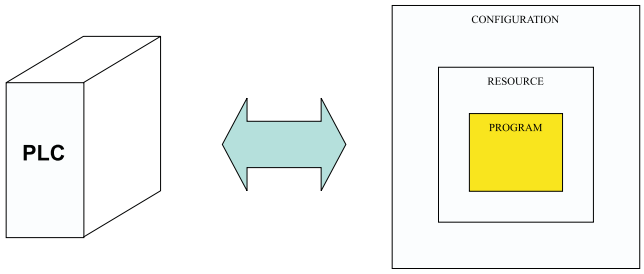



Figura 2.4: Sistema di controllo con PLC monoprocessore

Tuttavia, una situazione relativa a sistemi di dimensioni e complessità maggiori potrebbe necessitare di un maggior numero di controllori, connessi tra loro attraverso una rete di qualunque tipo, con diversi programmi per gestire sotto–parti del processo produttivo (vedi figura 2.5). In questo caso, benchè a livello "locale", cioè per uno specifico PLC la situazione è identica alla precedente, diventano fondamentali i costrutti di dichiarazione per variabili Access Path, che permettono di definire una zona di memoria dedicata allo scambio di informazioni tra i componenti della rete, con la sintassi descritta nell'esempio precedente.

Infine, una architettura ancora non molto comune, data la tecnologia attuale, potrebbe essere quella di un sistema basato su un unico PLC multi–processore, ed in questo caso il PLC consente certamente l'esecuzione contemporanea di diversi programmi, anche con tempistiche diverse (vedi figura 2.6), ognuno dei quali è "proprietario" di una determinata risorsa di esecuzione.

### 2.3.3 Program Organisation Units

Lo standard IEC definisce come Program Organisation Units (POU) i seguenti tre elementi del modello software: Programs, Function blocks e Functions.

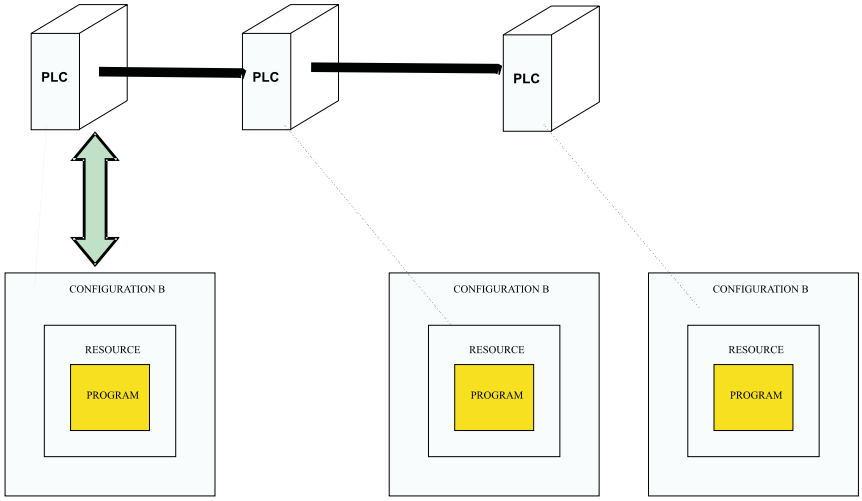



Figura 2.5: Sistema di controllo con più PLC connessi in rete

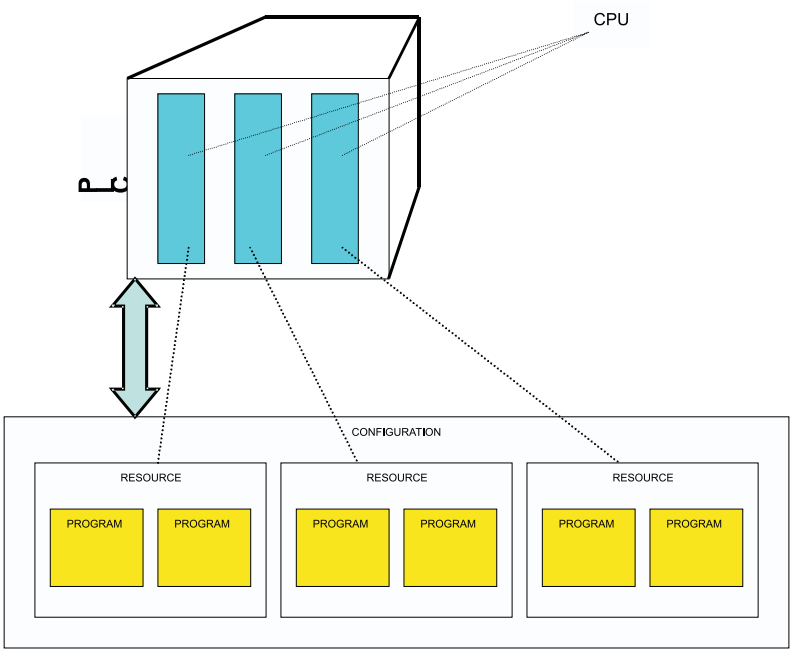



Figura 2.6: Sistema di controllo con un PLC multiprocessore

Questi tre oggetti rappresentano la parte certamente più interessante della norma IEC 1131-3, perchè definiscono finalmente una modalità di organizzazione dell'applicativo efficace ed orientata al riuso. Infatti, la proprietà comune di questi elementi software è che ognuno di essi può essere replicato in diverse parti di una applicazione, qualora le loro funzionalità debbano essere ripetute. Chiaramente la probabilità che ciò accada è maggiore quanto pià questa funzionalità è di basso livello gerarchico (cioè quanto più è "piccolo" il modulo).

Dato che lo standard punta molto, come detto, a fornire al programmatore gli strumenti per ottenere programmi modulari e ben organizzati gerarchicamente, le POU di livello inferiore (FB e Functions) rappresentano proprio gli elementi sui quali basare la fase di progettazione dell'applicativo.

I vantaggi derivanti dall'uso di queste POU si possono così riassumere:

- incremento di leggibilità del codice
- maggiore riutilizzabilità del codice
- fattorizzazione dei compiti in sottounità più semplici
- si evita la propagazione di errori sia in fase di stesura che di manutenzione
- suddivisione dell'applicativo in moduli il cui sviluppo può essere eseguito in parallelo da diverse persone.

**Definizione e uso di POU**

Una POU viene definita in modo generalizzato, associandole un nome (generico) che identifica un *tipo* di quella particolare POU, ma non ancora un vero e proprio modulo di programma eseguibile. Per poter inserire una POU, ad eccezione delle Function, in un applicativo essa va **istanziata**, cioè dichiarata tra i moduli effettivamente utilizzati assegnandole un nome simbolico specifico ed univoco. Ovviamente in questo modo possono essere utilizzate più istanze della stessa POU in un unico applicativo. L'istanza di una POU può, come fosse una qualunque variabile, essere dichiarata a qualunque livello internamente ad un'altra POU, cioè un Function Block può essere dichiarato a livello di program oppure "nascosto" all'interno di un'altro function block. Il motivo della necessità di istanziare una POU di tipo Program e Function Block risiede nella presenza di variabili locali di tipo statico, che devono mantenere il loro valore tra due cicli di esecuzione successivi. Istanziare la POU serve quindi a riservare un'area di memoria protetta per queste variabili, in una zona che non venga cancellata al termine del ciclo di programma. Normalmente questa area di memoria in un PLC viene detta **memoria di lavoro**.

In base alle definizioni dei diversi tipi di POU, fornite precedentemente, è quindi possibile identificare per ognuna di esse un ruolo ben preciso e darne perciò una finalità pratica:

| **Tipo di POU** | **Eseguito come:** | **Descrizione** |
|---|---|---|
| *Program* | Istanza di *program* | Consente il riutilizzo di software ad un macro livello |
| *Function Block* | Istanza di *function block* | Consente di riutilizzare algoritmi più o meno complessi (PID, filtri, ecc.) incapsulandone l'implementazione. |
| *Function* | Invocazione di *function* | Sono utilizzate per comuni manipolazioni di dati (funzioni matematiche, medie, ecc.). |

Una POU può, in linea di principio, essere programmata in uno qualunque dei linguaggi delle norma descritti nel seguito e contenere chiamate a qualunque altra POU istanziata. Lo standard vieta in modo esplicito solamente l'uso ricorsivo di POU: tale scelta è motivata dall'esigenza di determinismo delle prestazioni real-time dell'applicativo, nonchè da una semplificazione della fase di testing di quest'ultimo. Tali proprietà sarebbero messe in discussione se, ad esempio, una function richiamasse se stessa nella propria dichiarazione.

### 2.3.4 Dichiarazioni di variabili

Una caratteristica dello Standard che introduce funzionalità tipiche della programmazione di alto livello, è la possibilità di dichiarare di variabili all'interno di qualunque POU, con l'effetto di incapsulare e mascherare dati all'interno dei moduli software nei quali essi vengono utilizzati.

La sintassi IEC della dichiarazione di variabili consente di identificare immediatamente la loro visibilità e le modalità di accesso. Ad esempio, una variabile può essere strettamente locale, quindi accessibile solo all'interno della POU nella quale è dichiarata, oppure globale, e quindi visibile in tutti i blocchi di livello inferiore a quello che la dichiara. In quest'ultimo caso, tuttavia, la variabile può essere usata solo se espressamente indicata come variabile esterna, per evitare conflitti su nomi simbolici identici malauguratamente associati variabili logicamente diverse. Ad esempio, si supponga che la variabile `INT1`, intera, sia dichiarata globale a livello di program, e che debba essere utilizzata nell'FB1. La sintassi delle dichiarazioni è la seguente:

All'interno del PROGRAM:

```
VAR_GLOBAL
          INT1:INT;
END_VAR
```

All'interno del FB1

```
VAR_EXTERNAL
          INT1:INT;
END_VAR
```

Per quanto riguarda invece le dichiarazioni di variabili che non si intendono rendere visibili ad altre POU oltre a quella in cui sono dichiarate, nemmeno a POU che essa richiama, vi sono due possibili dichiarazioni, a seconda che si voglia che il valore delle variabili sia statico o temporaneo:

```
VAR
    nome_var1:tipo1;
    nome_var2:tipo2;
END_VAR
```

Costrutto per la dichiarazione di variabili interne, ovvero locali, utilizzabili solo nelle istruzioni di quella POU, non all'esterno e nemmeno nelle POU in essa richiamate.

```
VAR_TEMP
    nome_var1:tipo1;
    nome_var2:tipo2;
END_VAR
```

Costrutto per la dichiarazione di variabili interne temporeanee, utilizzabili come le precedenti, ma allocate in un area di memoria che viene cancellata al termine dell'esecuzione della POU. Anche se la POU viene richiamata più volte nello stesso ciclo di programma non è possibile recuperare il valore precedente.

Infine, vi sono delle variabili che assumono un ruolo fondamentale nella programmazione strutturata in moduli, in quanto permettono lo scambio di informazioni tra le varie POU in modo molto ben definito e controllabile dal programmatore. Esse sono i **parametri** delle POU, e possono essere di tre tipi, dichiarati con la sintassi seguente:

| | |
|---|---|
| `VAR_INPUT`<br>`nome_ingresso:tipo;`<br>`END_VAR` | Costrutto per la dichiarazione delle variabili di ingresso di una POU: contengono un valore fornito dall'esterno all'atto della chiamata della POU, accessibili in sola lettura. |
| `VAR_OUTPUT`<br>`nome_uscita:tipo;`<br>`END_VAR` | Costrutto per la dichiarazione delle variabili di uscita di una POU: il loro valore viene definito dalle istruzioni interne alla POU e fornito all'esterno, perciò sono accessibili anche in scrittura. Non è ammesso (nello standardil loro utilizzo per le Function, in quanto esse possono avere un unico risultato fornito all'esterno al termine della loro elaborazione, che viene associato per il chiamante con il nome stesso della Function. |
| `VAR_IN_OUT`<br>`nome_var:tipo;`<br>`END_VAR` | Costrutto per la dichiarazione delle variabili che possono fungere sia da ingresso che da uscita: in questo modo un accesso in scrittura all'interno della POU si ripercuote anche sul valore fornito dall'esterno. In pratica realizza una funzionalità analoga all'uso di variabili globali, permettendo a più POUs di modificare dei valori. |

Lo Standard IEC prevede, oltre ai parametri di ingresso e uscita definiti dall'utente, per Function e Function Block, due parametri predefiniti, che permettono di controllare l'esecuzione della POU. Questi parametri sono chiamati `EN`, ingresso della POU, ed `ENO`, uscita della POU. Come indica il nome, essi rappresentano una sorta di segnale di **enable** del blocco di programma, come per i dispositivi elettronici. Se questo segnale ha un valore logico falso, la POU non viene eseguita in nessun caso. Nel caso che questo ingresso non venga utilizzato, la POU viene comunque eseguita, dato che il suo valore di default è vero. Per quanto riguarda invece il segnale di uscita `ENO`, secondo lo Standard IEC esso assume un valore vero quando la POU ha terminato la propria esecuzione in modo corretto. Ad esempio, si supponga di avere una Function che esegue una somma tra valori interi. Nel caso che il risultato della somma ecceda il massimo valore rappresentabile dal dispositivo, il controllore normalmente segnala un errore di esecuzione al sistema operativo. In questo caso il valore di `ENO` rimane falso, pur essendo terminata l'esecuzione della Function.

Nella dichiarazione di variabili secondo lo Standard IEC è possibile inoltre fornire delle informazioni aggiuntive per il controllore, che permettono soprattutto di imporre un certo tipo di trattamento per la variabile, anche in relazione alla zona di memoria nella quale si richiede che essa sia memorizzata. Queste informazioni aggiuntive ed opzionali vengono dette *attributi*, compaiono nella dichiarazione a fianco della parola chiave `VAR(_...)`, e sono i seguenti:

- `RETAIN`: il valore della variabile deve essere memorizzata nella **memoria tamponata**[10], per essere mantenuto anche in caso di caduta dell'alimentazione;
- `CONSTANT`: il valore non può essere modificato in alcun modo (ovviamente non si può assegnare questo attribute a uscite, e secondo la Norma nemmeno a variabili esterne).
- `AT (indirizzo fisico)`: fissa una corrispondenza con una precisa locazione di memoria del PLC.

[10] memoria alimentata da una batteria indipendente dal modulo di alimentazione del PLC collegato alla fornitura elettrica dell'impianto industriale

In quest'ultimo caso l'indirizzo fisico deve essere espresso secondo la seguente sintassi:

- primo carattere: `%`
- secondo carattere: `I, O,` o `M`, rispettivamente per ingressi, uscite e memoria interna.
- (eventuale) terzo carattere: `X, B, W` o `L`, per indicare che si tratta, rispettivamente, di un bit, un byte, una word o una double-word.
- un campo numerico (o più campi separati da punti), identificativi della locazione fisica di memoria.

Questa sintassi può essere indicata anche all'interno di un'istruzione di programma, realizzando la modalità di **indirizzamento diretto**, "classica" dei programmi per PLC, utilizzando quelle che nella terminologia IEC sono dette Directly Represented Variables. Si noti che l'area di memoria interna accessibile direttamente viene detta **memoria di sistema**, e normalmente non corrisponde alla **memoria di lavoro** nella quale sono allocate le variabili locali delle POU, pertanto il loro uso non dovrebbe avere effetti collaterali sull'elaborazione del programma. Tuttavia, per ovvi motivi di portabilità e mantenibilità del programma se ne sconsiglia l'utilizzo, perlomeno nelle POU di livello gerarchico inferiore.

### 2.3.5 Tipizzazione dei dati

#### Tipi di dato

La variabili di un programma IEC devono essere forzatamente dichiarate preventivamente al loro utilizzo nel corso del programma. La dichiarazione richiede anche l'identificazione del tipo della variabile, in relazione a come essa è rappresentata in memoria. Un dato può essere:

- di tipo elementare.
- di tipo derivato.

I tipi elementari sono definiti nello standard in modo molto preciso, con riferimento al **nome** del tipo, alla sua **descrizione**, e alla sua **occupazione di memoria**. Questi tipi sono raggruppabili in quattro gruppi fondamentali:

- Booleani e stringhe di bit (`BOOL, BYTE, WORD,...`), per operazioni combinatorie;
- Numerici (`INT, LINT, REAL, LREAL,...`), per operazioni matematiche;
- Stringhe di caratteri (`STRING`), che possono contenere messaggi testuali;
- Valori temporali (`TIME, DATE, DATE_AND_TIME,...`).

Esiste poi la possibilità di definire variabili di tipo generico (`ANY, ANY_NUM, ANY_BIT,...`), allo scopo di implementare funzionalità adattabili a diversi tipi di variabili, realizzando il cosiddetto "overloading" degli operandi. Ad esempio, una funzione `MAX` che calcola il massimo tra più valori numerici, ha lo stesso tipo di funzionamento con interi e reali, perciò una sua definizione slegata dal tipo dei valori in ingresso ha un campo di applicabilità e riuso molto più vasto. Nella tabella 2.1 è rappresentata la gerarchia dei tipi di dato elementari e le loro generalizzazioni.

I tipi di dato derivato, invece, possono essere definiti dall'utente, grazie al costrutto:

```
TYPE
    nome_tipo_nuovo: tipo;
END_TYPE
```

Le modalità per definire un nuovo tipo di dato sono le seguenti:

```
ANY
+-ANY_DERIVED
+-ANY_ELEMENTARY
   +-ANY_MAGNITUDE
   |  +-ANY_NUM
   |  |   +-ANY_REAL
   |  |   |    LREAL
   |  |   |    REAL
   |  |   +-ANY_INT
   |  |        LINT, DINT, INT, SINT
   |  |        ULINT, UDINT, UINT, USINT
   |  +-TIME
   +-ANY_BIT
   |    LWORD, DWORD, WORD, BYTE, BOOL
   +-ANY_STRING
   |    STRING
   |    WSTRING
   +-ANY_DATE
        DATE_AND_TIME
        DATE, TIME_OF_DAY
```

Tabella 2.1: Gerarchia di generalizzazione dei tipi di dato

- definendo una struttura, cioè un insieme di variabili di tipo eterogeneo, raggruppati in un unico blocco di dati. Il costrutto sintattico è:

```
TYPE nome_tipo_nuovo
    STRUCT
          nome_var1: tipo1;
          nome_var2: tipo2;
    END_STRUCT;
END_TYPE
```

- elencandone espressamente i valori possibili (tipi di dato enumerativi), ad esempio:

```
TYPE Device_Mode:
     (INIT, RUNNING, STANDBY, FAULT);
END_TYPE
```

- fornendo un sottoinsieme dei valori possibili di un tipo di dato elementare, ad esempio un intervallo di valori numerici, definito con:

```
TYPE Temperature:INT(-50..400);
END_TYPE
```

- definendo un vettore, cioè una struttura contenente un certo numero di valori di tipo omogeneo, con il costrutto:

```
TYPE nome_tipo_nuovo:
     ARRAY [1..N] OF nome_tipo;
END_TYPE
```

## 2.4 Functions e Function Blocks predefiniti

Il ruolo principale delle POU come intese nello standard è quello di essere dei "contenitori" di codice riutilizzabile, una volta che questo sia stato testato opportunamente nello svolgimento della funzionalità prevista. Terminato il debug dei moduli, questi possono essere inseriti in diverse librerie che raggruppano POU con caratteristiche comuni. In una situazione ideale, l'utilizzatore finale degli strumenti della norma, vale a dire il programmatore, deve essere in grado di attingere il più possibile alle librerie pre–esistenti, i cui elementi vanno considerati come *scatole chiuse* che eseguono un compito ben determinato, ma il cui contenuto implementativo può essere ignorato senza che questo influenzi negativamente il lavoro del progettista.

A tale scopo, lo standard definisce un insieme di Functions e Function Blocks basilari che devono essere disponibili come componenti di libreria in ogni ambiente di programmazione, specificandone dettagliatamente interfaccia di ingresso–uscita e funzionalità. In tal modo si garantisce un set minimo di elementi di programma che il programmatore è certo di poter trovare in ogni tool di sviluppo, in modo che i programmi sviluppati utilizzando unicamente tali strumenti sono virtualmente portabili su ogni piattaforma. La libreria base ovviamente può essere integrata da librerie specifiche del fornitore del tool, o da librerie definite dall'utente stesso nel corso di progetti precedenti. Si noti che la norma non definisce l'implementazione interna dei componenti della libreria standard, in accordo con la loro interpretazione di "scatole chiuse", contenenti il know-how del produttore del tool applicato alla soluzione del compito richiesto.

Nel seguito vengono presentate le POU base definite nello standard, raggruppate per tipo di funzionalità eseguita.

### 2.4.1 Funzioni di conversione

La Norma prevede una forte tipizzazione dei dati. Inoltre, dato il suo contesto di applicazione, essa suggerisce che i controlli relativi alla consistenza di tipo negli assegnamenti e nelle manipolazioni delle variabili siano effettuati in modo statico in fase di compilazione. In questo modo il contenuto delle variabili è sempre trattato correttamente dal software programmato, in relazione al significato che la stringa di bit riferita dalla variabile assume secondo il tipo assegnatole. Lo scopo di questa filosofia di gestione dei tipi è quello di limitare il più possibile errori di programmazione che possano portare ad errori a tempo di esecuzione che possono compromettere il funzionamento (e di conseguenza anche l'integrità) dell'automatismo.

Secondo questa filosofia, il programmatore deve essere sempre consapevole del tipo di dato che viene elaborato in un certo set di istruzioni. Perciò eventuali conversioni di tipo devono essere eseguite esplicitamente con funzioni apposite, salvo nel caso di utilizzo del tipo di dato `ANY`, e non effettuate in modo implicito dal compilatore. In questa sezione sono quindi presentate le funzioni di conversione previste dalla norma, in relazione a tutti i tipi di dato che essa descrive.

#### Conversione generica

La sintassi delle funzioni di conversione esplicita di tipo è la seguente:

```
<Tipo di dato input>_TO_<Tipo di dato output>(Variabile_Input)
```

dove il prefisso ed il suffisso del nome della funzione indicano uno dei tipi definiti, mentre il valore di ritorno è quello della `Variabile_Input` convertito nel nuovo tipo. Ad esempio, data la seguente dichiarazione:

```
VAR
    INC: INT;
    RAMP_VAL: REAL;
    DISPLAY_VAL: STRING(16);
END_VAR
```

si possono effettuare le seguenti conversioni:

```
(* Conversione del valore intero in reale ed incremento del valore di
rampa *)

RAMP_VAL := RAMP_VAL + INT_TO_REAL(INC);

(* Conversione del valore reale in stringa, secondo la
rappresentazione numerica normale, per la visualizzazione su display
caratteri *)

DISPLAY_VAL := REAL_TO_STRING(RAMP_VAL);
```

Quest'ultima conversione verso stringa, trasforma un valore numerico nella sua rappresentazione a caratteri, ad esempio un valore 154.013 diventa la stringa `'154.013'`.

### Conversione di decimali in codifica binaria (BCD)

La norma definisce un insieme particolare di funzioni di conversione applicabili ai valori decimali in codifica binaria (**BCD**) contenuti in stringhe di bit di tipo `BYTE`, `WORD`, `DWORD`, `LWORD` verso tipi numerici interi (`SINT`,`INT`, `DINT`) e viceversa. Ad esempio:

```
VAR
    TEMP_INT: INT;
    PRESET: INT;
    DIFF: INT;
    BCD_TEMP_READ: WORD := 2#0010_0110_0011; (* BCD 611 *)
    BCD_OUT: WORD;
END_VAR

(* Conversione della word letta in intero, con valore di default 611*)
TEMP_INT := WORD_BCD_TO_INT(BCD_TEMP_READ);

(* Esecuzione di una differenza tra interi  *)
DIFF := PRESET - TEMP_INT;

(* Conversione in BCD del risultato *)
BCD_OUT := INT_TO_BCD_WORD(DIFF);
```

### Arrotondamento e troncamento di valori floating point (REAL)

La conversione di valori reali floating point in interi, ad esempio con una funzione `REAL_TO_INT()`, provoca un arrotondamento della parte decimale all'intero più vicino, in accordo con la norma IEC 559. Ad esempio, il valore 1.6 viene arrotondato a 3 e il valore negativo $-1.5$ convertito in $-2$.

Tuttavia, esiste una ulteriore modalità di conversione di valori reali floating point in valori interi, rappresentata dalla funzione di troncamento `TRUNC()`, eseguibile su una variabile di un qualunque tipo reale, che fornisce un valore assegnabile ad una variabile di un qualunque tipo intero. La parte di intestazione della funzione potrebbe quindi essere la seguente:

```
FUNCTION TRUNC: ANY_INT
    VAR_INPUT
           VAL: ANY_REAL;
    END_VAR
```

L'effetto della sua esecuzione consiste nell'eliminazione della parte decimale del valore in ingresso, restituendo come risultato la parte intera memorizzata nel formato opportuno. Ad esempio:

```
VAR
    MES_RPM: LREAL;
    PULSE: INT;
END_VAR

MES_RPM := 220.75;

(* Troncamento del valore reale *)
PULSE := TRUNC(MES_RPM);
```

In questo caso il valore assegnato a `PULSE` è 220, mentre con una conversione esplicita di tipo il valore sarebbe stato 221.

### 2.4.2 Funzioni aritmetiche

Le funzioni necessarie per effettuare elaborazioni di tipo numerico possono, secondo la norma, essere definite ed implementate dal produttore in modo "overloaded", cioè con un nome di funzione univoco che si "adatta" ad operare su molteplici tipi di dato, oppure in modo tipizzato, nel qual caso il nome della funzione deve fare riferimento al tipo di dati richiesti in input. Ad esempio, la funzione di nome `ADD`, per eseguire una somma, deve essere definita con parametri di tipo `ANY_NUM`, mentre la funzione di nome `ADD_INT` può ricevere solamente variabili `INT`. Occorre inoltre notare che una operazione aritmetica con variabili di diverso tipo, anche se effettuata con una funzione "overloaded" richiede le opportune conversioni di tipo esplicito, in modo che i tipi dei parametri attuali in ingresso siano compatibili con il tipo del valore di ritorno della funzione. Ad esempio:

```
VAR
   A: INT;
   B: INT;
   C: REAL;
   D: REAL
END_VAR

(* Conversione richiesta sul valore di A in ingresso alla funzione somma *)
C := INT_TO_REAL(A) + D;
```

```
(* Equivalente a: *)
    +-----------+   +-----+
A---|INT_TO_REAL|---|     |
    +-----------+   | ADD |---C
                  D-|     |
                    +-----+
```

```
(* Conversione richiesta sul valore di ritorno della funzione somma *)
D:= INT_TO_REAL(A + B);

(* Equivalente a: *)
   +-----+
A--|     |   +-----------+
   | ADD |---|INT_TO_REAL|--D
B--|     |   +-----------+
   +-----+
```

Alcune funzioni aritmetiche, corrispondenti a operatori commutativi, possono essere implementate con funzioni dette "estensibili", cioè che il cui numero di parametri di ingresso può essere esteso

indefinitamente. Ad esempio, proprio la funzione di somma è uno di questi operatori. Si hanno infatti le seguenti equivalenze:

```
D := A + B + C

(* Equivale a *)
   +---+
A--|   |   +---+
   |ADD|---|   |
B--|   |   |ADD|----D
   +---+   |   |
C----------|   |
           +---+

(* Che equivale all'estensione di ADD a tre ingressi *)
    +---+
A---|   |
    |   |
B---|ADD|----D
    |   |
C---|   |
    +---+
```

Nella tabella 2.2 sono elencate tutte le funzioni matematiche richieste dalla norma.

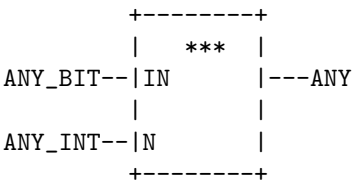
| **Nome funzione (simbolo)** | **Tipi I/O** | **Descrizione** |
|---|---|---|
| ABS | ANY_NUM | Valore assoluto |
| SQRT | ANY_REAL | Radice quadrata |
| LN | ANY_REAL | Logaritmo in base naturale ($e$) |
| LOG | ANY_REAL | Logaritmo in base 10 |
| EXP | ANY_REAL | Esponenziale naturale ($e^x$) |
| SIN | ANY_REAL | Funzione trigonometrica seno |
| COS | ANY_REAL | Funzione trigonometrica coseno |
| TAN | ANY_REAL | Funzione trigonometrica tangente |
| ASIN | ANY_REAL | Funzione trigonometrica arcoseno |
| ACOS | ANY_REAL | Funzione trigonometrica arcocoseno |
| ATAN | ANY_REAL | Funzione trigonometrica arcotangente |
| ADD (+) | ANY_MAGNITUDE | Funzione somma (**estensibile**) |
| MUL (*) | ANY_NUM | Funzione prodotto (**estensibile**) |
| SUB (-) | ANY_MAGNITUDE | Funzione sottrazione |
| DIV (/) | ANY_NUM | Funzione divisione. Con numeri interi fornisce il risultato della divisione con troncamento (es. $7/2 = 3$ e $(-7)/2 = -3$) |
| MOD | ANY_INT | Funzione modulo. Restituisce il resto della divisione tra interi, 0 se divisione per 0 |
| EXPT (**) | Base ANY_REAL, esponente ANY_INT, risultato ANY_REAL | Funzione di elevamento a potenza (Base$^{Esponente}$) |
| MOVE (:=) | ANY | Assegnamento, un ingresso e una uscita |

Tabella 2.2: Funzioni matematiche standard.

### 2.4.3 Funzioni booleane e per sringhe di bit

Le operazioni per le manipolazioni di stringhe di bit ed elaborazioni logiche sono ben espletate grazie alle funzioni standard IEC 1131-3. Quasi tutte le funzioni definite in questa sezione[11] sono "overloaded", cioè in grado di accettare un qualunque tipo di dato che sia una specificazione del tipo generico `ANY_BIT`, quindi `BOOL`, `BYTE`, `WORD`, `DWORD` o `LWORD`.

Le operazioni di rotazione e spostamento (shift) su stringhe di bit sono definite nella tabella 2.3, ed hanno tutte una interfaccia esterna con due valori di ingresso definita come segue:




| Nome funzione | Descrizione |
|---|---|
| SHL | Shift della stringa di bit di N posizioni verso sinistra, con inserimento di zeri nelle posizioni liberate a destra. |
| SHR | Shift della stringa di bit di N posizioni verso destra, con inserimento di zeri nelle posizioni liberate a sinistra. |
| ROR | Rotazione della stringa di bit di N posizioni verso destra. |
| ROL | Rotazione della stringa di bit di N posizioni verso sinistra. |

Tabella 2.3: Funzioni per stringhe di bit

Ad esempio, data la variabile `TEST: BYTE` contenente la stringa `2#0110_1010`, si hanno i seguenti risultati:

```
New_Byte1:= SHL(TEST, 4) (* valore 2#1010_0000 *)
New_Byte2:= SHR(TEST, 4) (* valore 2#0000_0110 *)
New_Byte3:= ROR(TEST, 3) (* valore 2#0100_1101 *)
New_Byte4:= ROL(TEST, 3) (* valore 2#0101_0011 *)
```

Le funzioni boolane definite nella norma sono anch'esse, come detto, applicabili ad ogni stringa di bit. Per valori di tipo `BOOL` permettono di effettuare le semplici operazioni di logica, molto spesso le operazioni più comuni in un programma per PLC. Applicate a stringhe di bit, eseguono l'operazione booleana tra coppie di bit in posizioni corrispondenti, determinando quindi il valore del bit nella stessa posizione nella stringa risultato. Tutte le funzioni definite nella tabella 2.4 sono estensibili, nel senso esposto nella sezione precedente. L'operazione di negazione, `NOT()`non è compresa nella tabella, dato che non possiede nè la caratteristica di estensibilità nè quella di essere "overloaded", richiedendo un ingresso di tipo `BOOL`.

| Nome funzione | Simbolo | Descrizione |
|---|---|---|
| AND | & | Prodotto logico. Il valore del risultato è vero se sono veri tutti gli ingressi. |
| OR | >=1 | Somma logica. Il valore del risultato è vero se è vero almeno uno degli ingressi. |
| XOR | =2K+1 | Or esclusivo. Corrisponde alla somma in complemento a 2, il risultato è vero se gli operandi sono complementari. |

Tabella 2.4: Funzioni booleane.

Si noti che i simboli definiti in questa tabella, ad eccezione del simbolo `&`, non possono essere utilizzati nei linguaggi testuali, ma solamente nell'identificazione di blocchi grafici.

[11] con l'eccezione della negazione `NOT`

### 2.4.4 Funzioni di selezione

Le funzioni di selezione predefinite dallo standard permettono di scegliere tra un certo numero di valori quello corrispondente ad un determinato criterio. Ad esempio, il criterio può essere "il massimo valore", oppure l'indirizzamento di un certo segnale tra diversi multiplexati. Queste funzioni possono essere usate con qualunque tipo di dato, perciò sono "overloaded" al massimo livello di genericità (tipo di dato `ANY`). Si veda la tabella 2.5

| Nome funzione | Ingressi | Descrizione |
|---|---|---|
| `SEL` | `G: BOOL;`<br>`IN0: ANY;`<br>`IN1: ANY;` | **Selezione**: se `G` è vero il risultato equivale `IN1`, altrimenti equivale `IN0` |
| `MAX` | `IN0: ANY_ELEMENTARY;`<br>`IN1: ANY_ELEMENTARY;`<br>(estensibile) | **Massimo**: il risultato equivale il massimo fra i valori in ingresso |
| `MIN` | `IN0: ANY_ELEMENTARY;`<br>`IN1: ANY_ELEMENTARY;`<br>(estensibile) | **Minimo**: il risultato equivale il minimo fra i valori in ingresso |
| `LIMIT` | `MN: ANY_ELEMENTARY;`<br>`IN: ANY_ELEMENTARY;`<br>`MX: ANY_ELEMENTARY;` | **Limite**: il risultato equivale `IN` se è compreso tra `MN` e `MX`, altrimenti equivale il più prossimo di questi ultimi. |
| `MUX` | `K: ANY_INT;`<br>`IN0: ANY;`<br>`IN1: ANY;`<br>(estensibile) | **Multiplexer**: il risultato equivale all'ingresso indicizzato da `K`, ad esempio se `K` vale 0 seleziona `IN0`, se 1 `IN1` ecc. |

Tabella 2.5: Funzioni di selezione

Si noti che la norma prevede la tipizzazione della funzione `MUX`, tramite l'indicazione del tipo dell'indice e dei tipi dei valori selezionati. Ad esempio:

```
FUNCTION MUX_INT_REAL: REAL
        VAR_INPUT
             K: INT;
             IN0: REAL;
             IN1: REAL;
             ...
        END_VAR
```

### 2.4.5 Funzioni di comparazione

Le funzioni predefinite di comparazione tra valori dello stesso tipo (tutte le funzioni in queste sezione sono definite con ingressi `ANY_ELEMENTARY`) sono utili per costruire espressioni condizionali (v. costrutti `IF..THEN` o cicli iterativi nel linguaggio *Structured Text*, o condizioni di transizioni nel linguaggio *Sequential Function Chart*). Tali funzioni eseguono le operazioni di confronto più comuni nei linguaggi di programmazione di alto livello: uguaglianza, disuguaglianza, maggiore o minore di, ecc. (v. tabella 2.6), fornendo un risultato booleano (vero o falso). Le funzioni, eccezion fatta per il test di disuguaglianza, sono estensibili a molteplici ingressi, con il significato che lo stesso test viene effettuato sugli ingressi considerati nella loro sequenza, ed il risultato equivale all'esecuzione dell'operazione `AND` tra tutti i test. Ad esempio:

```
(* Controllo se i quattro ingressi hanno valori crescenti *)
ControlOK := GT(In4, In3, In2, In1);

(* Equivale a *)
ControlOK := In4 > In3 & In3 > In2 & In2 > In1;
```

| Nome funzione | Simbolo | Descrizione |
|---|---|---|
| GT | > | **Maggiore di**: risultato vero se il valore del primo ingresso è maggiore del secondo (**estensibile**). |
| GE | >= | **Maggiore o uguale di**: risultato vero se il valore del primo ingresso è maggiore del secondo oppure se i due sono uguali (**estensibile**). |
| EQ | = | **Uguale**: risultato vero se gli ingressi sono uguali tra loro (**estensibile**). |
| LT | < | **Minore di**: risultato vero se il valore del primo ingresso è minore del secondo (**estensibile**). |
| LE | <= | **Minore di**: risultato vero se il valore del primo ingresso è minore del secondo oppure se i due sono uguali (**estensibile**). |
| NE | <> | **Disuguaglianza**: risultato vero se il valori dei *due* ingressi sono diversi (**non estensibile**). |

Tabella 2.6: Funzioni di comparazione.

Si noti che il simboli corrispondenti ai nomi delle funzioni possono essere usati come operatori nei linguaggi testuali, contrariamente a quelli definiti nella tabella 2.4.

### 2.4.6 Funzioni di manipolazione di stringhe di caratteri

La Norma definisce una vasta gamma di funzioni definite espressamente per la manipolazione di stringhe di caratteri, utili per generare messaggi testuali da inviare a interfaccie uomo–macchina (**MMI**) oppure per interpretare comunicazioni provenienti da e dirette ad altri dispositivi di comunicazione, anche remoti. Si noti che le stringhe di caratteri rappresentano un tipo di dato molto "ingombrante" per dispositivi dotati di memoria limitata come i Controllori Programmabili. Solitamente il costruttore definisce una dimensione massima per le stringhe, un'aspetto non coperto dallo Standard. Tuttavia, tale dimensione va tenuta in considerazione per evitare errori a run–time, ad esempio in caso di concatenazione di più stringhe, nel qual caso i messaggi risultanti potrebbero essere troncati. La tabella 2.7 contiene l'elenco completo delle funzioni predefinite per le stringhe.

### 2.4.7 Funzioni per valori temporali e tipi enumerati

In relazione alle operazioni eseguibili su tipi di dato che esprimono un valore temporale, cioè `TIME`, `DATE`, `TIME_OF_DAY` e `DATE_AND_TIME`, la prima edizione della norma prevede che possano essere eseguite anche con le funzioni matematiche "overloaded" descritte nella sezione 2.4.2. Tuttavia, data la particolarità del tipo di dato, che richiede una modalità di memorizzazione e di implementazione delle funzioni fortemente dipendente dall'hardware, è stata espressamente affermata l'intenzione di richiedere nelle future versioni dello standard la tipizzazione stretta delle funzioni di elaborazione matematica su valori temporali.

Di conseguenza, tali funzioni devono avere una sintassi del tipo `<Operazione>_<Tipo primo operando>_<Tipo secondo operando>`, come ad esempio `ADD_TOD_TIME` o `SUB_DT_TIME`. Le operazioni di moltiplicazione e divisione possono essere eseguite solamente tra un operando di tipo `TIME` e uno di tipo `ANY_NUM` con le funzioni `MULTIME` e `DIVTIME`.

Inoltre, lo standard richiede le funzioni di conversione `DT_TO_TOD` e `DT_TO_DATE`, che hanno l'effetto di estrarre rispettivamente la parte relativa al tempo del giorno e alla data di un valore

| **Nome funzione (tipo di ritorno)** | **Ingressi** | **Descrizione** |
|---|---|---|
| `LEFT` (ANY_STRING) | `In: ANY_STRING;`<br>`L: ANY_INT;` | Estrae gli `L` caratteri più a sinistra nella stringa `In`. |
| `RIGHT` (ANY_STRING) | `In: ANY_STRING;`<br>`R: ANY_INT;` | Estrae gli `R` caratteri più a destra nella stringa `In`. |
| `MID` (ANY_STRING) | `In: ANY_STRING;`<br>`L: ANY_INT;`<br>`R: ANY_INT;` | Estrae i caratteri della stringa partendo dall'`L`–esimo da sinistra al `R`–esimo da destra nella stringa `In`. |
| `CONCAT` (ANY_STRING) | `In1: ANY_STRING;`<br>`In2: ANY_STRING;`<br>(estensibile) | Fornisce una stringa contenente tutti i caratteri delle due (o più) stringhe in ingresso, concatenate una di seguito l'altra. |
| `INSERT` (ANY_STRING) | `In1: ANY_STRING;`<br>`In2: ANY_STRING;`<br>`P: ANY_INT;` | Inserisce i caratteri della stringa `In2` nella stringa `In1`, partendo dalla posizione `P`–esima dall'inizio (sinistra) di `In1`. |
| `DELETE` (ANY_STRING) | `In: ANY_STRING;`<br>`L: ANY_INT;`<br>`P: ANY_INT;` | Cancella `L` caratteri dalla stringa `In`, partendo dalla posizione `P`–esima da sinistra. |
| `REPLACE` (ANY_STRING) | `In1: ANY_STRING;`<br>`In2: ANY_STRING;`<br>`L: ANY_INT;`<br>`P: ANY_INT;` | Sostituisce `L` caratteri di `In1` prelevandoli da `In2`, partendo dalla posizione `P`–esima da sinistra di `In1`. |
| `FIND` (ANY_INT) | `In1: ANY_STRING;`<br>`In2: ANY_STRING;` | Restituisce la posizione da sinistra della stringa `In2` all'interno della stringa `In1`, se essa è contenuta, 0 altrimenti. |
| `LEN` (ANY_INT) | `In: ANY_STRING;` | Fornisce la lunghezza della stringa `In`. |

Tabella 2.7: Funzioni di manipolazione di stringhe di caratteri.

DATE_AND_TIME. La funzione di concatenazione CONCAT_DATE_TOD, invece, ha l'effetto di unire una data ad un valore TIME_OF_DAY per ottenere un valore DATE_AND_TIME.

Infine, la norma prevede la possibilità di applicare alcune funzioni standard anche a tipi non predefiniti, ma definiti dall'utente con la modalità enumerativa (ad es. TYPE MODE_TYPE:(RUNNING,STOPPED,STANDBY) END_TYPE). Queste funzioni sono: SEL, MUX, EQ (=) e NE (<>).

### 2.4.8 Function Blocks standard

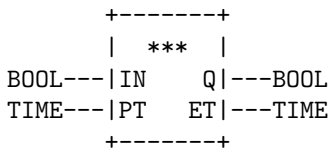
L'elemento del modello software IEC 1131-3 *Function Block* come definito nella sezione da spec. permette di realizzare librerie di strumenti molto più potenti e complessi che non quelli forniti dalle *Functions* descritte nelle sezioni precedenti. La maggiore potenza degli FB è dovuta alla loro capacità di registrare i propri dati interni in variabili locali, il cui valore viene mantenuto fra un ciclo di esecuzione dell'algoritmo di controllo e il successivo. Inoltre gli FB possono fornire diversi valori di uscita, che contengono una potere informativo notevolmente superiore rispetto al singolo valore di ritorno della funzione.

La Norma definisce un certo numero di FB standard derivanti dallo studio degli strumenti maggiormente richiesti ed utilizzati nei programmi di controllo di automatismi. Tra questi, rivestono un ruolo determinante i **contatori** ed i **temporizzatori**, tanto che, per migliorare l'efficienza di esecuzione dei programmi, essi sono implementati in routine di basso livello (firmware) o addirittura direttamente nell'hardware nella maggior parte dei PLC.

#### Temporizzatori

I temporizzatori sono molto utilizzati nella pratica per controllare la corretta attivazione di determinati segnali, ritardando o prolungando la durata dell'impulso fornito in uscita dal dispositivo controllore. Ad esempio, potrebbe essere necessario attivare, con un opportuno ritardo dal rilevamento di un prodotto tramite fotocellula, il dispositivo elettromeccanico di apertura di un fermo, per consentire il passaggio del prodotto senza che questo intralci altri meccanismi. Oppure, un certo azionamento pneumatico potrebbe richiedere un'attivazione prolungata per un certo periodo di tempo per permettere il corretto posizionamento di un pezzo meccanico. I periodi di tempo richiesti per il corretto funzionamento dell'automatismo devono essere rispettati con precisione dal programma di controllo, e questo è possibile grazie all'uso dei temporizzatori, tramite i quali, come detto, è possibile accedere a clock di sistema ad alta risoluzione.

I temporizzatori definiti nella norma hanno la seguente interfaccia:

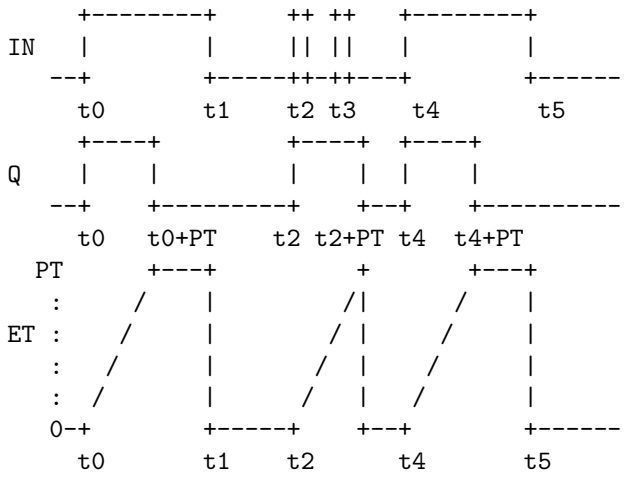



nella quale l'ingresso PT è il valore di tempo impostato, IN è l'attivazione del conteggio, ET è il tempo trascorso dall'inizio del conteggio e Q è un valore logico vero o falso che dipende dal tipo di temporizzatore, nel modo descritto nel seguito:

- **Pulse timer** (TP), temporizzatore ad impulso.

  Il conteggio comincia al fronte di salita di IN, l'uscita Q rimane alta per tutto il tempo del conteggio. Solo terminato il conteggio è possibile rilevare un nuovo fronte di salita dell'attivazione.

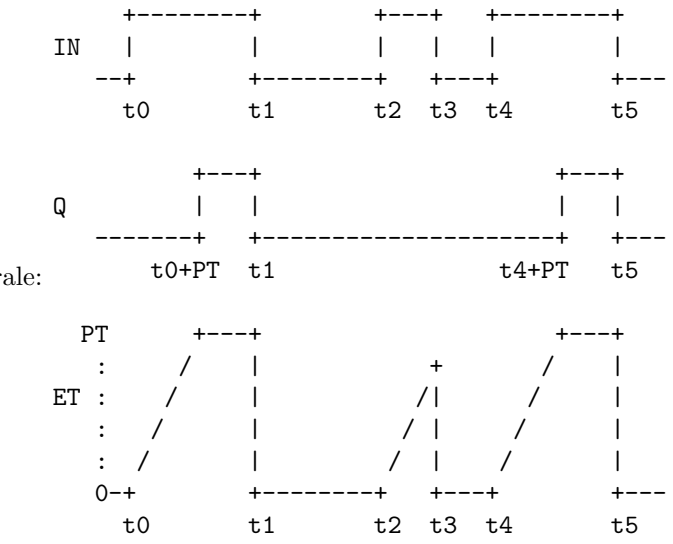
  Diagramma temporale:




- **On–delay timer**, (`TON`) temporizzatore con ritardo all'inserimento.

  Il conteggio inizia con il fronte di salita dell'attivazione e l'uscita logica assume valore vero al termine del conteggio solamente se il segnale di attivazione è rimasto sempre vero.

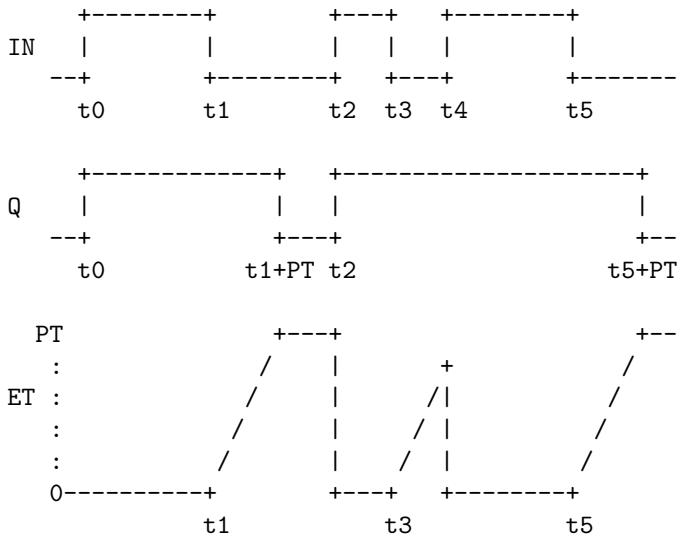
Diagramma temporale:




- **Off–delay timer**, (`TOF`) temporizzatore con ritardo al disinserimento.

  Il conteggio inizia con il fronte di discesa dell'attivazione, e l'uscita logica è vera finchè è vera l'attivazione e, dopo il fronte di discesa di quest'ultima, finchè non è terminato il conteggio.

  Diagramma temporale:




### Contatori

I contatori sono anch'essi blocchi molto importanti, in quanto permettono di effettuare in modo molto semplice il rilevamento di un prefissato numero di impulsi su un certo segnale, senza costringere il programmatore a scrivere una routine per la gestione dell'accumulatore e i relativi test. Ad esempio, un contatore permette di attivare il ciclo di chiusura di una scatola quando essa è stata riempita con il numero di prodotti richiesto e, dato che valore di conteggio è un parametro dell'FB relativo, questo può essere impostato tramite una interfaccia operatore, semplificando la riconfigurazione della macchina.

I contatori standard sono essenzialmente di tre tipi:

- **Up Counter**, contatori in avanti.

```
       +-----+
       | CTU |
BOOL--->CU  Q|---BOOL
BOOL---|R    |
 INT---|PV CV|---INT
       +-----+
```

  L'accumulatore `CV` viene incrementato di una unità ad ogni fronte di salita di `CU`, e l'uscita `Q` è vera se esso supera il valore impostato in `PV`. Un valore vero del segnale `R` azzera il valore dell'accumulatore. Si noti che il nome `CTU` indica un contatore per tipi di dato `INT` . La norma richiede che le versioni associate ai tipi `DINT`, `LINT`, `UDINT` e `ULINT` siano indicate con il nome `CTU_<Tipo>`.

- **Down Counter**, contatore all'indietro.

```
       +-----+
       | CTD |
BOOL--->CU  Q|---BOOL
BOOL---|LD   |
 INT---|PV CV|---INT
       +-----+
```

  In questo caso il contatore esegue un decremento di una unità del valore di conteggio `CV` sul fronte di saltia di `CU`, e l'uscita `Q` è vera quando il contatore arriva a 0. Il contatore deve essere

preimpostato con un valore vero su `LD`, che ha l'effetto di assegnare il valore di `PV` al valore di conteggio. Anche per il nome `CTD` la norma prevede che esso indichi il contatore con valore `INT`, e `CTD_<Tipo>` i contatori con valori degli altri tipi interi.

- **Up–Down Counter**, contatore in avanti e all'indietro.

```
        +------+
        | CTUD |
BOOL--->CU  QU|---BOOL
BOOL--->CD  QD|---BOOL
BOOL---|R     |
BOOL---|LD    |
 INT---|PV  CV|---INT
        +------+
```

  Questo contatore bivalente racchiude entrambe le funzionalità dei contatori descritti precedentemente. Tuttavia, in caso di contemporanea richiesta di conteggio in avanti e all'indietro, viene data la precedenza alla prima. Come per gli altri contatori è richiesta la tipizzazione esplicita nel nome del FB, altrimenti si intende il funzionamento con il tipo `INT`.

### 2.4.9 Altri blocchi utili

Funzionalità molto comuni nei programmi di controllo per PLC sono quelle di riconoscimento di fronti di salita e discesa su segnali digitali e di memorizzazione di valori logici tramite elementi bistabili (blocchi di Set–Reset). Anche tali funzionalità sono state standardizzate dalla norma, che richiede per esse l'implementazione dei blocchi descritti nella tabella 2.8.

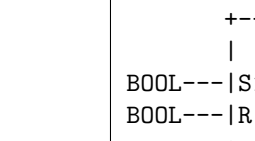
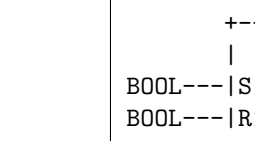
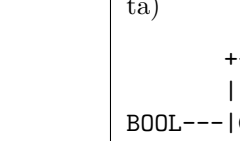
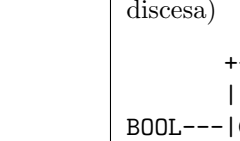
| Nome Function Block | Descrizione |
|---|---|
| **Bistabile SR** (Set dominante)<br>  | Un valore vero di `S1` rende vera l'uscita `Q1`, se non lo è già. L'uscita viene resettata solo se è vero `R` e non `S1`. |
| **Bistabile RS** (Reset dominante)<br>  | Un valore vero di `S` e nel contempo falso di `R1` rende vera l'uscita `Q1`, se non lo è già. L'uscita viene resettata se è vero `R1`. |
| **Rising–Edge Trigger** (fronte di salita)<br>  | L'uscita `Q` assume valore vero per un solo ciclo di esecuzione se viene rilevato un fronte di salita sul `CLK`, cioè se esso è passato da 0 a 1 tra il ciclo precedente e quello attuale. |
| **Falling–Edge Trigger** (fronte di discesa)<br>  | L'uscita `Q` assume valore vero per un solo ciclo di esecuzione se viene rilevato un fronte di discesa sul `CLK`, cioè se esso è passato da 1 a 0 tra il ciclo precedente e quello attuale. |

Tabella 2.8: Function Blocks Trigger e Bistabili.

# Capitolo 3

# Lo Standard IEC 61131-3: i linguaggi di programmazione

## 3.1 Introduzione

Come detto nel capitolo 2, un aspetto dei controllori industriali tradizionalmente ostile per il progettista è la difformità dei linguaggi di programmazione. In tal senso, la Norma intende regolamentare l'aspetto puramente sintattico dei linguaggi di programmazione di controllori industriali, definendo quattro linguaggi standard dalle caratteristiche differenti, che possono coprire tutte le necessità del progettista nello sviluppo di un programma di controllo. Grazie alla standardizzazione della sintassi dei linguaggi, il compito di programmazione risulta semplificato anche in caso di utilizzo di hardware fornito da differenti produttori, con una conseguente notevole riduzione dei tempi di sviluppo del software di controllo e di conseguenza del "time to market" della macchina.

I linguaggi definiti nella norma IEC 61131-3 si dividono in due categorie: *testuali* e *grafici*. Nel seguito vengono descritti i due linguaggi testuali Instruction List (IL) (sezione 3.2) e Structured Text (ST) (sezione 3.3), mentre le sezioni successive 3.4 e 3.5 descrivono rispettivamente il linguaggio grafico Ladder Diagram (LD) e il linguaggio grafico Function Block Diagram (FBD).

## 3.2 Instruction List (IL)

La tipica applicazione dei PLC industriali consiste nello svolgere compiti di controllo logico con operazione prevalentemente booleane e l'architettura hardware delle loro CPU è ottimizzata per svolgere questo tipo di operazioni in modo semplice e rapido. I linguaggi di livello assemblativo di tali dispositivi sono pertanto basati storicamente su una sintassi del tipo **1 operatore : 1 operando**. Con questa sintassi, occorre un particolare tipo di registro interno del processore, detto *accumulatore*, nel quale memorizzare il risultato dell'operazione, mentre l'eventuale secondo operando necessario all'operazione (es. somma logica tra due valori booleani) consiste implicitamente nel contenuto dell'accumulatore stesso.

Anche lo standard IEC 61131-3, essendo derivato dall'analisi dettagliata del panorama corrente del mercato PLC, segue questo tipo di sintassi per la definizione dell'Instruction List (IL), il linguaggio testuale di livello inferiore nello standard. Essendo un linguaggio assemblativo, tutti gli altri linguaggi possono avere un'equivalente in IL, mentre non è sempre possibile convertire codice IL in altri linguaggi. Infatti, l'IL si presta per sua natura alla generazione di codice ottimizzato, il quale tuttavia potrebbe perdere l'equivalenza con codice di alto livello o diagrammi grafici in LD o FBD.

### 3.2.1 Sintassi dell'IL

Seguendo la terminologia della Norma, una riga di codice IL si compone di una eventuale **label** (etichetta) seguita da **:**, un **operatore**, un'eventuale **modificatore** ed un **operando**. Ad esempio:

```
START:  LD   \%IX0.1    (* if Button-1 *)
        ANDN \%MX1.5    (* and not Inhibition *)
        ST   \%QX3.2    (* Light-3 on *)
```

Una istruzione IL si considera terminata con il termine della riga testuale, quindi i commenti debbono terminare prima del termine della riga stessa. Inoltre i commenti non possono precedere l'istruzione, perciò devono comparire nella parte destra della riga.

Nell'esempio precedente si possono identificare la label `START`, gli operatori `LD`, `AND` e `ST` ed il modificatore `N` applicato all'operatore `AND`. Come detto, una riga di codice `OP Operando` va interpretata implicitamente come: `Accumulatore := Accumulatore OP Operando`. La prima istruzione della sequenza carica (`LoaD`) nell'accumulatore il contenuto del bit di ingresso all'indirizzo 0.1 (acceduta direttamente), la seconda ne esegue l'`AND` logico con il bit di memoria `%MX1.5` negato (tale è il significato del modificatore `N`) ed infine il risultato del prodotto logica tra le due variabili è viene assegnato al bit di uscita `%QX3.2`, grazie all'istruzione `ST` (`STore`).

Un altro possibile modificatore di istruzioni è la parentesi tonda, che permette di differire l'esecuzione dell'istruzione che precede l'apertura della parentesi fino al termine delle istruzioni racchiuse tra parentesi. Il risultato finale dell'esecuzione di queste ultime rappresenta l'operando per l'istruzione differita. Ad esempio:

```
AND( \%IX0.1
OR   \%MX3.5
)
```

equivale a `Accumulatore := Accumulatore AND (%IX0.1 OR %MX3.5)`. Si noti che la variabile `%IX0.1` deve essere caricata nell'accumulatore per poter effettuare l'operazione successiva. Questo significa che il valore precedente dell'accumulatore deve essere salvato opportunamente (es. su uno "stack") e poi recuperato per l'esecuzione dell'operazione differita. La Norma accetta anche la forma in cui il caricamento della prima variabile dopo l'apertura di parentesi è esplicitato, nel modo seguente:

```
AND(
LD   \%IX0.1
OR   \%MX3.5
)
```

Questa forma permette di eseguire anche operazioni diverse, ad es. chiamare una funzione o caricare il valore negato della variabile (`LDN`), nella posizione immediatamente successiva all'apertura di una parentesi.

Nella tabella 3.1 sono descritti tutti gli operatori definiti nella Norma. I tipi di dato ammessi per gli operandi sono intesi secondo quanto espresso nei paragrafi..., mentre il valore del risultato è da intendersi compatibile con quello dell'operando con l'eccezione delle istruzioni di comparazione, per le quali il risultato si intende sempre booleano (`BOOL`).

### 3.2.2 Chiamate di Functions e Function Blocks in IL

La sintassi delle chiamate di altre POUs in codice IL può assumere diverse forme. Per quanto riguarda le funzioni, esse possono essere invocate direttamente, senza precedere il nome della funzione con un operatore. Infatti, il nome di una funzione può essere considerato esso stesso un operatore, dato che si ottiene un unico risultato in uscita dall'esecuzione della Function, il quale è automaticamente memorizzato nell'accumulatore. Il nome della funzione è seguito dalla lista delle assegnazioni ai parametri formali, la quale può essere esplicita o implicita. Ad esempio, la forma esplicita:

```
LIMIT(
EN:= COND,
IN:= B,
MN:= 1,
```

| Operatore | Modificatore ammesso | Significato |
|---|---|---|
| LD | N | Carica il valore dell'operando nell'accumulatore. |
| ST | N | Assegna il valore dell'accumulatore alla variabile operando. |
| S | Nessuno | Pone l'operando al valore booleano vero (1) se l'accumulatore ha un valore booleano vero, altrimenti lo lascia invariato. L'operando deve essere di tipo BOOL. |
| R | Nessuno | Pone l'operando al valore booleano falso (0) se l'accumulatore ha un valore booleano vero, altrimenti lo lascia invariato. L'operando deve essere di tipo BOOL. |
| AND o & | N, ( | Esegue il prodotto logico fra l'operando e l'accumulatore. |
| OR | N, ( | Esegue la somma logica fra l'operando e l'accumulatore. |
| XOR | N, ( | Effettua l'operazione di OR esclusivo tra l'operando e l'accumulatore. |
| NOT | Nessuno | Effettua la negazione logica (complemento a uno) dell'accumulatore. Non richiede operando. |
| ADD | ( | Somma matematica tra operando e accumulatore. |
| SUB | ( | Sottrazione matematica dell'operando dall'accumulatore. |
| MUL | ( | Moltiplica l'accumulatore per l'operando. |
| DIV | ( | Divide l'accumulatore per l'operando. |
| MOD | ( | Effettua il modulo, cioè calcola il resto della divisione tra accumulatore e operando. |
| GT | ( | Confronta se l'accumulatore è maggiore dell'operando. |
| GE | ( | Confronta se l'accumulatore è maggiore o uguale all'operando. |
| EQ | ( | Confronta se l'accumulatore è uguale all'operando. |
| NE | ( | Confronta se l'accumulatore è diverso dall'operando. |
| LE | ( | Confronta se l'accumulatore è minore o uguale all'operando. |
| LT | ( | Confronta se l'accumulatore è minore dell'operando. |
| JMP | C, N | Salta alla label indicata nell'operando. Se modificato da C il salto è condizionato dal valore boolano vero dell'accumulatore, se modificato con C ed N è condizionato dal valore boolano falso dell'accumulatore. |
| CAL | C, N | Chiama il Function Block indicato nell'operando. I modificatori C ed N agiscono come indicato in precedenza. |
| RET | C, N | Forza l'uscita dalla Function, dal Function Block o dal Program attuale. I modificatori C ed N agiscono come indicato in precedenza. Non richiede operando. |
| ) | Nessuno | Termina una sequenza iniziata con una istruzione modificata da ( e fa eseguire l'operazione differita. |

Tabella 3.1: Operatori del linguaggio Instruction List

```
MX:= 5,
ENO=> TEMP
)
ST A
```

non richiede di seguire un ordine preciso nell'elenco dei parametri, dato che le assegnazioni non sono ambigue. Al contrario, nell'invocazione:

```
LIMIT B,1,5
ST A
```

occorre seguire attentamente l'ordine previsto in sede di definizione della funzione.

Nel caso di chiamate di Function Block, il linguaggio IL prevede un'istruzione speficica, `CAL`, che deve precedere il nome dell'istanza di FB. Tuttavia, l'assegnazione dei parametri formali può avvenire in altri modi, oltre al metodo con lista di assegnazioni (esplicite o implicite) precedentemente descritto per le Functions. Infatti, si può assegnare ai vari parametri il contenuto dell'accumulatore, tramite l'istruzione `ST` ed in seguito, una volta effettuata l'assegnazione di tutti i parametri necessari, è possibile eseguire la chiamata del FB. Ad esempio:

```
VAR
    IT_PHCELL: BOOL AT \%IX0.2;
    ITEMS:= INT;
    COUNT_ITEM: CTU;
END_VAR

LD 50
ST COUNT_ITEM.PV     (* Imposta il contatore a 50 pezzi *)
LD IT_PHCELL
ST COUNT_ITEM.CU     (* Assegna il segnale da conteggiare *)
CAL COUNT_ITEM       (* Chiama il contatore *)
```

In questo esempio, viene utilizzato un FB standard definito dalla Norma. Per questi FB, è possibile sfruttare un'ulteriore metodo di assegnazione dei parametri e di chiamata, utilizzando il nome del parametro del FB come un operatore, rendendo implicita l'istruzione `ST`. Di conseguenza, il codice precedente è equivalente a:

```
LD 50
PV COUNT_ITEM        (* Imposta il contatore a 50 pezzi *)
LD IT_PHCELL
CU COUNT_ITEM        (* Assegna il segnale ed effettua il conteggio *)
```

Si noti l'assenza dell'istruzione `CAL`. Secondo la descrizione fornita dalla Norma, il parametro `CU` inteso come contatore rende implicita anche l'istruzione di chiamata. Questo si spiega con la necessità di rilevare i fronti di salita del segnale da contare al fine di effettuare correttamente il conteggio, per cui effettuare una chiamata del blocco senza assegnare tale parametro impedisce di rilevarne le variazioni. Con la chiamata implicita tale "errore" non può essere commesso.

### 3.2.3 Commenti e critiche

Sebbene questo aspetto non sia trattato esaustivamente nello Standard, il contenuto dell'accumulatore e l'operando di una istruzione IL devono ovviamente essere compatibili per tipo tra di loro e con l'operatore, secondo la filosofia seguita dalla Norma in relazione alla tipizzazione dei dati. Alcune critiche sulla definizione dell'IL vertono proprio sull'assenza di indicazioni sufficientemente dettagliate soprattutto per quanto riguarda gli errori da rilevare, come riconoscere tali errori in fase di compilazione, oppure quale comportamento deve avere il controllore se gli errori sono rilevati a "run–time", indicazioni necessarie per poter implementare correttamente un compilatore o una "macchina virtuale" in grado di interpretare il linguaggio IL.

## 3.3 Structured Text (ST)

Il linguaggio testuale Structured Text (ST) rappresenta il passaggio alla programmazione strutturata e di alto livello per i controllori industriali, pur mantenendo la semplicità e la chiarezza sintattica necessarie per essere utilizzato con successo nello sviluppo di applicazioni anche complesse, ma basate su dispositivi dalle prestazioni computazionali limitate.

Il linguaggio ST si presta per la risoluzione in modo rapido ed intuitivo di qualsiasi tipo di compito di programmazione. Tuttavia, dato il campo di applicazione ed il contesto nel quale è definito, considerando cioè anche gli altri linguaggi grafici o testuali della Norma IEC 61131-3, risulta particolarmente adatto per eseguire complesse elaborazioni matematiche, realizzabili con poche righe di codice, oppure nel caso in cui occorra eseguire test condizionali con molteplici alternative, i quali richiederebbero con la logica a contatti o con il linguaggio IL un grande numero di operazioni di salto con la conseguente riduzione di leggibilità del software sviluppato.

### 3.3.1 Espressioni ed assegnazioni nel ST

Nel codice ST, assumono una notevole importanza la valutazione di `espressioni`, con conseguente `assegnazione` del valore del risultato ad una variabile. L'istruzione di assegnazione ha la seguente sintassi:

```
Variabile := Espressione;
```

L'espressione è composta da `operatori` ed `operandi` e viene valutata secondo le regole di precedenza definite nella Norma, che assegnano ad ogni operatore un certo livello di priorità. A parità di priorità, si considera prima l'operatore più a destra. Ad esempio:

```
VAR
   A, B, C, D, RES: INT;
END_VAR

A := 1;
B := 2;
C := 3;
D := -4;

RES := A+B-C*ABS(D);  (* RES assume il valore -9 *)

(* Applicando le parentesi, operatore con la massima priorita',
   il risultato assegnato a RES diventa 0 *)

RES := (A+B-C)*ABS(D);
```

Le semplici istruzioni dell'esempio precedente evidenziano le potenzialità del ST rispetto ad altri linguaggi, ad esempio l'IL. Con quest'ultimo, sarebbero infatti necessarie ben più di una riga di codice (9 righe nel primo caso, 7 nel secondo, rispettando le priorità).

Anche espressioni booleane sono semplici da implementare con l'ST. Ad esempio, date le stesse variabili dell'esempio precedente con le stesse assegnazioni, l'espressione:

```
(A>B)&(C<D)
```

ha come risultato finale il valore booleano falso.

Nella tabella 3.2 sono indicati tutti gli operatori ammessi in una espressone ST, con il rispettivo livello di priorità.

Si noti che la chiamata di una qualunque funzione, standard o definita dall'utente, deve sempre essere parte di un'espressione, in modo che il suo risultato sia assegnato correttamente ad una variabile del programma o come valore intermedio di un'espressione complessa. Gli FB invece, possono essere chiamati semplicemente con una istruzione del tipo:

| Simbolo | Operazione | Priorità |
|---|---|---|
| `( )` | | **MAGGIORE** |
| `fun(args)` | Valutazione di funzione | |
| `-` | Negazione (matematica) | |
| `NOT` | Negazione (booleana) | |
| `**` | Elevamento a potenza | |
| `*` | Moltiplicazione | |
| `/` | Divisione | |
| `MOD` | Modulo | |
| `+` | Somma | |
| `-` | Sottrazione | |
| `<, <=, >, >=` | Test comparativi | |
| `=` | Uguaglianza (test) | |
| `<>` | Disuguaglianza | |
| `&` o `AND` | AND booleano | |
| `XOR` | OR booleano esclusivo | |
| `OR` | OR booleano | **MINORE** |

Tabella 3.2: Operatori per espressioni dello Structured Text

```
FB_nome_istanza(Par1 := A, Par2:=B ...);
```

e i suoi parametri di output accessibili in qualunque istruzione con la sintassi `FB_nome_istanza.Out1`.

### 3.3.2 Costrutti di selezione e di iterazione

La definizione del ST contiene la descrizione dei costrutti tipici dei linguaggi di programmazione di alto livello, in particolare per la selezione di alternative e la ripetizione iterata di codice.

I costrutti di `selezione` sono due: `IF..THEN..ELSE` e `CASE..OF`. Nel primo si ha la valutazione di una espressione booleana che permette di selezionare fra due alternative a seconda del suo risultato, mentre nel secondo si ha la valutazione di una espressione che può avere un risultato di tipo `ANY_INT` oppure di tipo definito dall'utente in modo enumerativo, perciò è possibile selezionare fra molteplici alternative corrispondenti ad un diverso valore assunto dall'espressione. Nell'esempio seguente sono descritte due implementazioni della stessa selezione utilizzando i due diversi costrutti, sfruttando tutte le parole chiave definite per essi dalla Norma:

```
VAR
    Sel : INT;
END_VAR

(* Selezione con IF *)
IF Sel = 0 THEN
            istruzioni0;
ELSIF Sel = 1 THEN
             istruzioni1;
   ELSIF Sel = 2 THEN
               istruzioni2;
      ELSE istruzionielse;
END_IF;

(*Selezione con CASE *)
CASE Sel OF
   0: istruzioni0;
   1: istruzioni1;
```

```
   2: istruzioni2;
ELSE istruzionielse;
END_CASE;
```

In questo caso, ovviamente, la soluzione con `CASE` risulta più compatta e leggibile, ma il costrutto `IF` vanta comunque una maggiore flessibilità grazie alla possibilità di costruire espressioni booleane complesse per eseguire la selezione, combinando diversi tipi di operatori di test e operatori logici.

I costrutti di `iterazione` sono invece tre:

- ```
  WHILE Espressione_booleana DO
  Istruzioni
  ...;
  END_WHILE
  ```
- ```
  REPEAT
  Istruzioni
  ..;
  UNTIL Espressione_booleana
  END_UNTIL
  ```
- ```
  FOR Variabile_intera := Valore_Iniziale TO Valore_Finale BY
  Passo_di_incremento
  DO Istruzioni;
  ...
  END_FOR
  ```

In ogni caso, si tratta costrutti che permettono di eseguire pivolte, nella stessa scansione del programma, una determinata sequenza di istruzioni, fino a quando una certa condizione non sia verificata. I tipi di iterazione si distinguono tra loro in base al momento in cui viene effettuata la verifica della condizione o anche al tipo di condizione stessa. Ad esempio, la scansione dello stato di 10 diversi dispositivi, con interruzione della scansione in caso di stato negativo, può essere effettuata nei seguenti modi:

```
VAR
   Count : INT;
   Dev_Status: BOOL := TRUE;
END_VAR

Count := 0;
WHILE Count < 10 & Dev_Status  DO
      Dev_Status := Check_Device(Dev_number := Count, ...);
      Count := Count+1;
END_WHILE;

(* Equivale a: *)
Count := 0;
REPEAT
      Dev_Status := Check_Device(Dev_number := Count, ...);
      Count := Count+1;
UNTIL Count >= 10 OR NOT(Dev_Status)
END_REPEAT;

(* Equivale a: *)
FOR Count := 0 TO 9 BY 1 DO
      Dev_Status := Check_Device(Dev_number := Count, ...);
      IF NOT(Dev_Status) THEN EXIT;
```
```

```
    END_IF;
END_FOR;
```

Si noti la differenza dell'espressione booleana nel test dei costrutti `WHILE` e `REPEAT` per implementare la stessa funzionalità nel caso che il test sia eseguito rispettivamente prima o dopo l'esecuzione del codice da ripetere. Il costrutto `FOR` invece, integra nell'intestazione `FOR Count := 0 TO 9 BY 1 DO` l'indicazione esatta del numero di iterazioni da effettuare, richiedendo all'utente di specificare per una determinata variabile intera il valore iniziale, finale ed il passo di incremento ad ogni ciclo[1]. Tuttavia, il test di iterazione implicito non può contenere altri controlli, perciò occorre effettuare la necessaria verifica dello stato del dispositivo all'interno del codice ripetuto ed eventualmente forzare l'uscita anticipata dal ciclo. Questo è possibile nel linguaggio ST grazie all'istruzione `EXIT`.

Ovviamente, date le esigenze di *real–time* del software di controllo, l'implementazione di cicli con i costrutti descritti richiede una grande attenzione da parte del programmatore, soprattutto per quanto riguarda la descrizione delle condizioni di uscita dal ciclo: occorre valutare attentamente il numero massimo di iterazioni che è possibile eseguire in una singola scansione di programma, le situazioni che richiedono un'interruzione immediata delle iterazioni ed in base a questo determinare quale costrutto sia più idoneo al caso specifico.

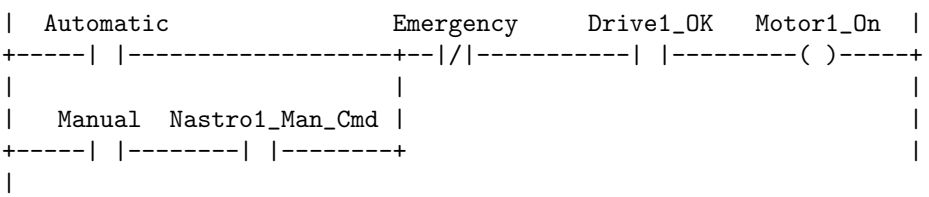
## 3.4 Ladder Diagram (LD)

Considerazioni analoghe a quelle fatte sulla definizione dell'Instruction List possono essere fatte sul linguaggio grafico Ladder Diagram (LD). Infatti, anche questo linguaggio si basa sull'osservazione delle caratteristiche comuni tra le varie implementazioni della logica di programmazione maggiormente diffuza tra gli utenti di PLC: la logica a contatti o *relay ladder logic*. Tali caratteristiche comuni sono pertanto unificate nella definizione del LD, in modo da permettere al programmatore di avere una approccio più immediato ai diversi editor grafici di differenti produttori.

Il concetto alla base del LD è quello di rappresentare graficamente un flusso virtuale di corrente elettrica tra due barre di potenziale, regolato da interruttori e bobine, in modo da implementare in modo molto intuitivo una logica booleana: passaggio di corrente = `TRUE`, assenza di corrente = `FALSE`. Una riga di codice corrisponde perciò ad una rete (**network**) di contatti, Functions, Function Blocks e bobine, connesse da linee (eventualmente estese da etichette testuali dette *connettori*, in caso di suddivisione su più righe o pagine) attraversate dall'ideale flusso di corrente. La Norma stabilisce che il flusso di corrente si intende diretto da destra a sinistra e che le network vanno valutate (eseguite) dall'alto verso il basso, a meno che non siano presenti istruzioni di salto o di termine forzato dell'esecuzione della POU.

I vari tipi di contatto e bobina utilizzabili in una network LD sono descritti nella tabella 3.3.

Una semplice network in LD che ne evidenzia l'orientamento alla realizzazione di operazioni combinatorie potrebbe essere la seguente:




con la quale viene controllata l'attivazione un determinato motore, ad esempio per movimentare un nastro trasportatore. Tale attivazione è condizionata in ogni caso dall'assenza di situazioni di emergenza e dall'abilitazione del relativo drive del motore (AND logico tra le due). Se la macchina controllata è in funzionamento automatico non vi sono altre condizioni da controllare, mentre se la

[1] La parola chiave `BY` è opzionale, se non presente viene assunto il passo di incremento unitario, perciò nell'esempio essa è ridondante

| Simbolo | Descrizione |
|---|---|
| `***`<br>`--\| \|--` | **Contatto normalmente aperto**<br>La corrente passa verso destra solamente se il valore della variabile booleana indicata da `***` è vero. |
| `***`<br>`--\|/\|--` | **Contatto normalmente chiuso**<br>La corrente passa verso destra solamente se il valore della variabile booleana indicata da `***` è falso. |
| `***`<br>`--\|P\|--` | **Contatto rilevatore di fronti di salita**<br>La corrente passa verso destra solamente se il valore della variabile booleana indicata da `***` è passato da un valore falso nella scansione precedente del programma ad un valore vero in quella attuale. |
| `***`<br>`--\|N\|--` | **Contatto rilevatore di fronte di discesa**<br>La corrente passa verso destra solamente se il valore della variabile booleana indicata da `***` è passato da un valore vero nella scansione precedente del programma ad un valore falso in quella attuale. |
| `***`<br>`--( )--` | **Bobina momentanea**<br>Se l'espressione booleana a sinistra della bobina ha valore vero la variabile booleana `***` assume valore vero per la durata della scansione corrente del programma. |
| `***`<br>`--( )--` | **Bobina momentanea negata**<br>Se l'espressione booleana a sinistra della bobina ha valore vero la variabile booleana `***` assume valore falso per la durata della scansione corrente del programma. |
| `***`<br>`--(S)--` | **Bobina con memoria**<br>Se l'espressione booleana a sinistra della bobina ha valore vero la variabile booleana `***` assume valore vero e lo mantiene fino ad un esplicito *reset* da parte di una bobina con memoria negativa. |
| `***`<br>`--(R)--` | **Bobina con memoria negativa**<br>Se l'espressione booleana a sinistra della bobina ha valore vero la variabile booleana `***` assume valore falso e lo mantiene fina ad un esplicito *set* da parte di una bobina con memoria. |
| `***`<br>`--(P)--` | **Bobina per fronti di salita**<br>Se l'espressione booleana a sinistra della bobina passa da un valore falso nella precedente scansione del programma ad un valore vero in quella attuale la variabile booleana `***` assume valore vero (solo per la durata della scansione corrente). |
| `***`<br>`--(N)--` | **Bobina per fronti di discesa**<br>Se l'espressione booleana a sinistra della bobina passa da un valore vero nella precedente scansione del programma ad un valore falso in quella attuale la variabile booleana `***` assume valore vero (solo per la durata della scansione corrente). |

Tabella 3.3: Contatti e bobine del Ladder Diagram.

macchina è in funzionamento manuale, occorre che l'operatore attivi uno specifico comando tramite console per il movimento di quel determinato nastro: l'OR tra la condizione di "Automatico" quella di manuale "Manuale" viene realizzato dalla congiunzione tra i due rami parallali a sinistra nella network, infatti per avere il flusso virtuale di corrente è sufficiente che essa scorre in almeno uno di essi.

### 3.4.1 Connessioni, salti e terminazione di POUs in LD

La Norma non definisce una lunghezza massima per una network LD, ma la lascia come parametro dipendente dal fornitore dell'ambiente di programmazione. Tuttavia, essa si rivela molto importante, in quanto la programmazione tipica del linguaggio richiede molto spesso lunghe sequenze per il controllo di numerose condizioni necessarie all'attivazione di determinate uscite, condizioni che permettono di tenere traccia dello stato di funzionamento dell'automatismo in modo esplicito (anche se talvolta di difficile interpretazione e leggibilità). Viceversa, vengono definiti in modo molto preciso i simboli di connessione tra network suddivise su più righe di codice:

```
...--->Etichetta_A>


>Etichetta_A>---...
```

Si noti che la connessione descritta modifica solamente l'editazione del programma e non affligge in alcun modo il flusso di esecuzione delle istruzioni.

Molto diverso è invece il significato del simbolo che identifica l'istruzione di salto in LD, sebbene esso sia graficamente simile a quello precedentemente descritto. Infatti, grazie all'istruzione di salto il programmatore può controllare completamente il flusso di esecuzione, forzando l'esecuzione anticipata di codice, ad esempio in relazione al rilevamento di malfunzionamenti, oppure impedendo l'esecuzione di determinate istruzioni in certo stato dell'automatismo. Le istruzioni di salto vengono eseguite se, come le bobine, vi è un flusso di corrente nella network. In pratica, il salto è condizionato dal valore dell'espressione booleana nel ramo alla sinistra dell'istruzione di salto. Graficamente, i salti in LD si indicano come segue:

```
|
+------->>Etichetta_B      (* Salto incondizionato *)
|
...

Etichetta_B:
|
+-----...
|

...

|  Esegui_Salto
+-----| |---------->>Etichetta_C   (* Salto condizionato *)
|
...

Etichetta_C:
|
+----...
|
```

Si noti che la Norma impone che l'etichetta che identifica l'obiettivo del salto debba essere contenuta nella stessa POU dell'istruzione di salto. Qualora il codice LD descriva il contenuto di un'azione in un SFC, il limite di validità del salto è l'azione stessa. Tuttavia, la norma non vieta che l'etichetta preceda l'istruzione di salto stessa, permettendo in pratica la realizzazione di cicli iterativi analoghi a quelli descritti nella sezione 3.3.2. Ovviamente, valgono le stesse esigenze di accortezza ivi descritte in relazione alle esigenze di determinismo richieste dal software real–time.

Un'altra istruzione che modifica il flusso di esecuzione delle network in LD è quella di uscita da una POU e ritorno al chiamante. Tale istruzione ha una sintassi molto simile a quella dell'istruzione di salto ed ha l'effetto di interrompere l'esecuzione delle istruzioni contenute nella POU, in modo che l'elaborazione del programma prosegua dall'istruzione immediatamente successiva a quella di chiamata della POU attuale. Come per l'istruzione di salto, l'istruzione di ritorno deve essere il simbolo terminale di una network e può quindi essere un ritorno condizionato o incondizionato. Si noti che una POU programmata in LD può terminare con l'istruzione di ritorno incondizionato, in alternativa alle espressioni chiave `END_FUNCTION`, `END_FUNCTION_BLOCK` o `END_PROGRAM`. Graficamente, si rappresenta come segue:

```
|   Alarm_Cond
+------| |---------<RETURN>    (* Ritorno condizionato *)
|

...

|
+------<RETURN>       (* Termine della POU *)
|
```

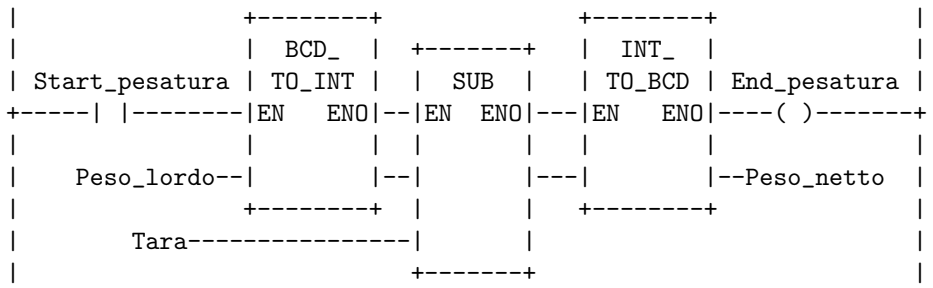
### 3.4.2 Functions e Function Blocks in LD

Con le sole istruzioni "native" del linguaggio LD non sarebbe possibile eseguire elaborazioni su variabili di altro tipo se non booleane. Pertanto per eseguire altre operazioni occorre inserire nelle network LD delle chiamate di Functions o FB, utilizzandone la rappresentazione grafica descritta nella sezione 2.3. L'inserimento di blocchi grafici in LD deve comunque rispettare il principio base del linguaggio, vale a dire il controllo del flusso di corrente (booleano) tra le barre di potenziale. Anche per questo motivo la Norma prevede le variabili booleane predefinite `EN` ed `ENO` per tutte le Functions e gli FB. Ad esempio, una semplice chiamata incondizionata di un FB, il quale potrebbe anche non avere variabili di ingresso uscita booleane, ha la seguente sintassi:

```
|               FB_Nome
|         +-----------------+          |
+---------|EN   FB_Tipo  ENO|----------+
|         |                 |          |
|   Var1--|Intero1    Result|--Finale  |
|         |                 |          |
|   Var2--|Reale1           |          |
|         |                 |
|         +-----------------+
```

Risulta evidente che il semplice inserimento di contatti sul ramo alla destra della variabile `EN` del FB, trasforma l'istruzione in una chiamata condizionata del blocco.

Elaborazioni maggiormente complesse, ad esempio espressioni matematiche, possono richiedere l'inserimento di numerosi blocchi grafici in una stessa network. Inoltre, alcuni di questi blocchi possono richiedere in ingresso dei risultati forniti in uscita da altri blocchi. In questo caso, si possono utilizzare variabili temporanee per memorizzare tali risultati intermedi, oppure connettere graficamente in modo diretto i blocchi. Ad esempio:

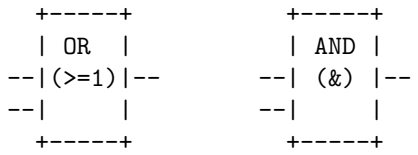



Nella network precedente viene elaborato un segnale proveniente da un sensore di peso, che fornisce un valore codificato in BCD. Per poter determinare il peso netto, il valore ricevuto deve essere convertito in intero, prima di sottrarre la tara (un valore intero prefissato). Di conseguenza, occorre la funzione `BCD_TO_INT`, il cui valore di ritorno viene collegato direttamente all'ingresso relativo al primo operando dell'operazione di sottrazione. Il valore restituito da quest'ultima viene poi nuovamente convertito in BCD per poter essere inviato direttamente ad un display per la visualizzazione.

## 3.5 Function Block Diagram (FBD)

Il linguaggio grafico Function Block Diagram è un formalismo molto efficace e descrittivo, grazie al quale l'algoritmo di controllo definito tramite il flusso di segnali attraverso una rete di blocchi grafici interconnessi tra loro, secondo uno schema molto simile ai diagrammi che descrivono i circuiti elettronici. Questi blocchi grafici elaborano i segnali collegati ai loro parametri di ingresso e trasmettono i risultati dell'elaborazione attraverso i connettori connessi ai loro parametri di uscita.

I blocchi utilizzabili per la realizzazione dell'elaborazione corrispondono alla rappresentazione grafica di tutte le Functions e i Function Blocks predefiniti descritti nella sezione 2.4. Ad esempio, per valutare una somma o un prodotto logico tra due variabili booleane, occorrono i blocchi:




Oltre a questi, possono essere inseriti i blocchi realizzati dall'utente. Proprio questo aspetto rende il formalismo particolarmente interessante, anche dal punto di vista della documentazione del programma. Infatti, esso permette di descrivere in maniera molto semplice ed immediata tutti gli scambi di informazioni tra i moduli di controllo programmati per determinati compiti, e permette di verificare anche visivamente come e quando essi interagiscono nel corso dell'elaborazione del programma.

Gli aspetti sintattici di maggiore interesse del linguaggio FBD, al di là delle convenzioni sull'aspetto dei connettori e dei blocchi, del resto abbastanza intuitive, riguardano prevalentemente l'orientamento del **flusso di segnale** e le regole per la **valutazione complessiva della rete**. Per quanto riguarda il flusso di segnale esso si considera, analogamente al flusso di corrente virtuale del Ladder Diagram, diretto da sinistra verso destra. In relazione alla valutazione degli elementi di una rete FBD, vi sono tre principi fondamentali:

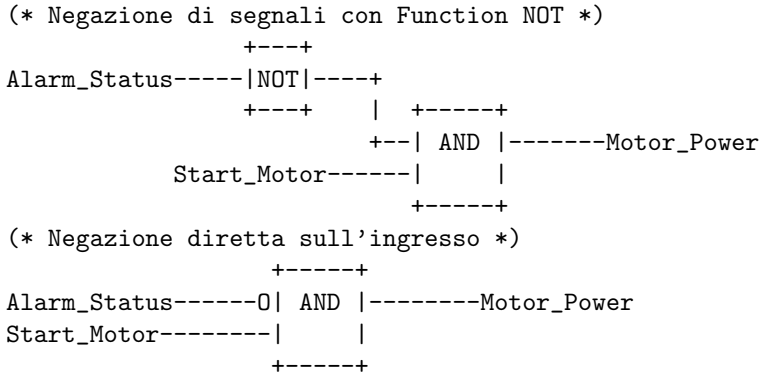
1. Nessun elemento della rete deve essere valutato prima che siano stati valutati i valori di tutti i propri ingressi.
2. La valutazione di un elemento della rete non è completa finchè non sono stati valutati i valori di tutte le sue uscite.

3. La valutazione della rete termina quando tutte le uscite di tutti i suoi elementi sono state valutate.

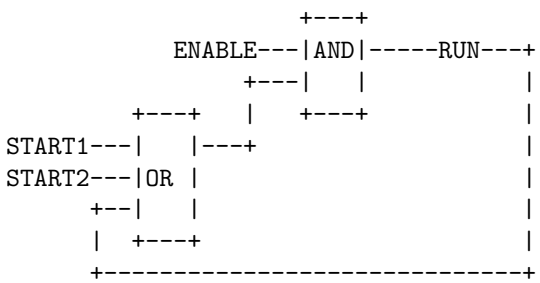
Nonostante queste regole, alcuni particolari rimangono dipendenti dall'implementazione. Ad esempio, mentre per il linguaggio LD è espressamente richiesto che le network separate siano valutate dalla prima in alto all'ultima in basso, questo non è specificato per le reti FBD separate tra loro. Tuttavia, è prassi comune da parte dei realizzatori di ambienti di programmazione aderenti allo Standard, seguire la stessa convenzione del Ladder Diagram, cioè effettuare la valutazione delle reti separate partendo dalla prima in alto nel diagramma.

### 3.5.1 Diagrammi FBD con blocchi standard

Essendo il formalismo FBD fortemente autoesplicativo, risulta utile analizzare degli esempi di diagrammi per approfondire la conoscenza del linguaggio. Nell'esempio seguente, sono rappresentati due modi di effettuare la negazione di segnali booleani consentiti dalla Norma IEC 61131-3:



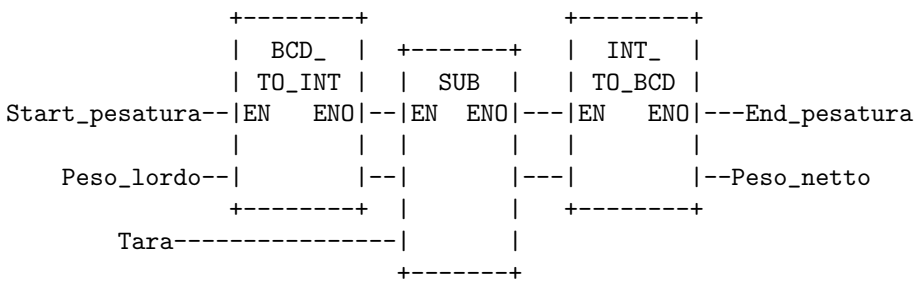

Un aspetto molto interessante del FBD è la possibilità di esplicitare dei percorsi di retroazione per segnali, detti **feedback paths**. Ad esempio:




In questo caso, le regole di valutazione della rete non sono applicabili in maniera diretta, perciò lo Standard impone che il valore del segnale retroazionato sia considerato pari al valore di default della variabile (nell'esempio `RUN`) se definito, altrimenti pari al valore di default del tipo della variabile (nell'esempio la variabile `RUN` è di tipo booleano, perciò il suo valore di default è `FALSE`), ed il valore che assume al termine della valutazione della rete sia utilizzato come valore iniziale per la valutazione nel ciclo di esecuzione successivo del programma.

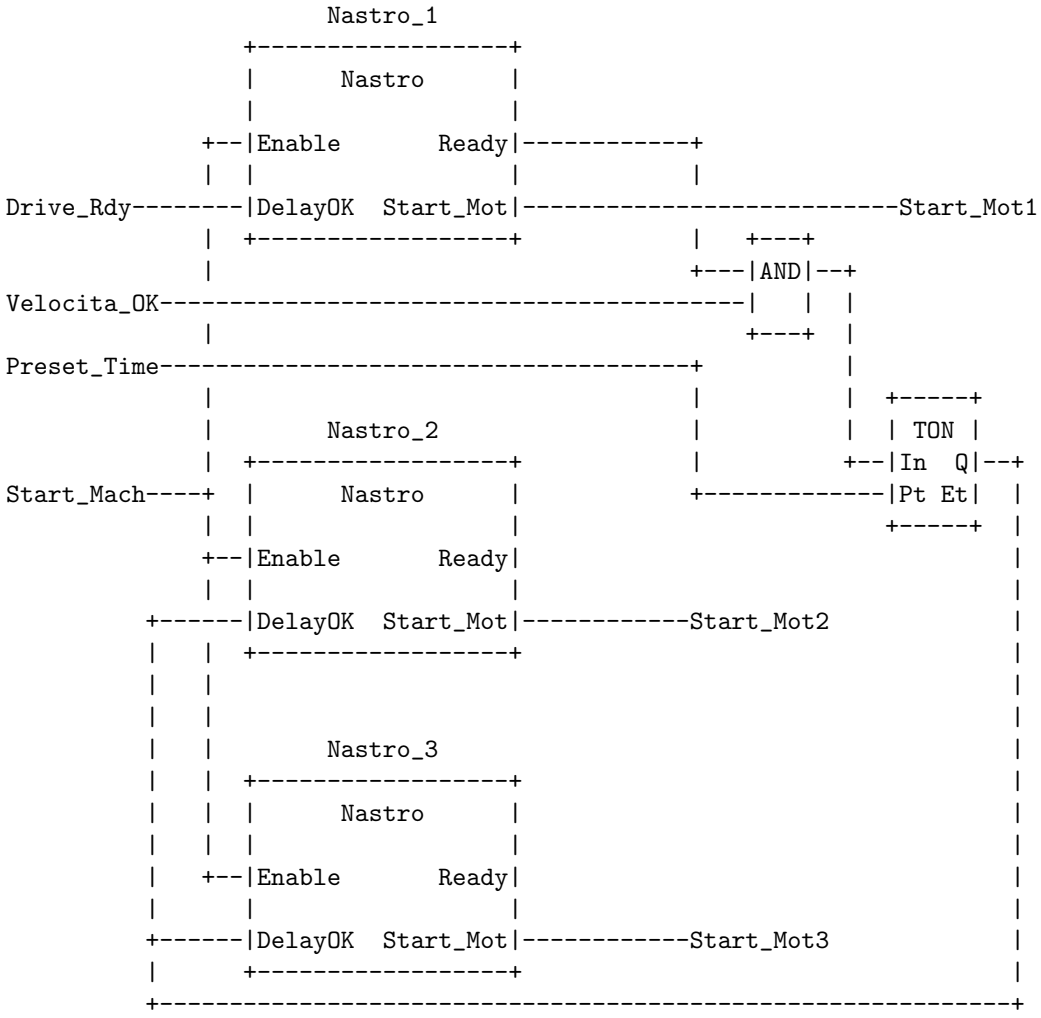
I parametri predefiniti `EN` ed `ENO` diventano un valido strumento per esercitare un controllo esplicito della valutazione di una rete FBD, nel caso che non siano sufficientemente deterministiche le tre regole esposte precedentemente, oppure si voglia prevenire l'esecuzione di operazioni indesiderate. Come detto nel capitolo 2, un valore booleano falso sull'ingresso `EN` inibisce l'esecuzione della Function o del Function Block, mentre l'uscita `ENO` assume un valore vero solamente quando il blocco ha terminato la sua esecuzione ed ha restituito il controllo alla POU chiamante. Un esempio di utilizzo di

tali parametri era stato proposto nella sezione 3.4.2, pertanto nel seguito si ne mostra la realizzazione in FBD:




### 3.5.2 Descrizione tramite FBD delle relazioni tra blocchi di programma

Dagli esempi precedenti si evince come il FBD sia semplice ed intuitivo per la programmazione di POU anche sfruttando solamente Functions e Function Block standard. Tuttavia, come detto nel paragrafo introduttivo, esso si rivela notevolmente efficace per la descrizione delle interrelazioni tra i Function Blocks programmati dall'utente, diventando anche un interessante strumento di documentazione. Ad esempio, si consideri una cella di trasporto prodotti in una macchina automatica. Essa risulta composta da un certo numero di nastri movimentati da motori elettrici, anche di diverso tipo e potenza. Tuttavia, dal punto di vista del controllo logico da parte del PLC, tali nastri possono spesso essere considerati equivalenti, pertanto ognuno di essi può essere associato ad una istanza di un Function Block di tipo *"Nastro"*. Nell'esempio proposto nel seguito, si considera una cella di trasporto costituita da tre nastri, attivati alla partenza della macchina secondo una certa sequenza per assicurare che l'ultimo di questi nastri abbia raggiunto una certa velocità di regime prime che gli altri si muovano. La rete FBD che descrive il funzionamento della cella è la seguente:


Nastro_1
Nastro
Enable
Ready
Drive_Rdy
DelayOK
Start_Mot
Start_Mot1
AND
Velocita_OK
Preset_Time
Nastro_2
TON
In
Q
Pt
Et
Start_Mach
Start_Mot2
Nastro_3
Start_Mot3

# Bibliografia

[1] "Software di sistema per S7-300/400, Sviluppo di programmi, Manuale di Programmazione", Manuale Siemens, Cod. C79000-G7072-C506-01.

[2] IEC 61131-3 (1993-03) "Programmable controllers - Part 3: Programming languages", International Electrotechnical Commission (1993).

[3] Robert W. Lewis "Programming Industrial Control Systems using IEC 61131-3", Revised Edition, Institution of Electrical Engineers (1998).