EZ-Red - Modulo I/O di potenza

Manuale di programmazione

Altri manuali sono disponibili al sito http://www.xonelectronics.it

Indice

Introduzione

Riepilogo hardware

EZ-Red è un modulo di interfaccia di potenza a 24 volt, che si collega con l'interfaccia USB a un computer per controllare un processo esterno. Il modulo dispone di:

- 8 ingressi digitali, più 2 veloci - optoisolati, con interfaccia encoder (opzionale)
- 8 uscite di potenza a 24 volt
- 2 ingressi analogici 0-10V, e 2 uscite analogiche 0-10V
- Watch-dog interno, con disposizione uscite configurabile
- Memoria flash (non volatile) per memorizzare il ciclo del PLC e variabili di configurazione

Utilizzi del ciclo PLC

EZ-Red è in grado di controllare un ciclo operativo anche senza essere collegato a un computer, in modo autonomo. Per fare questo, occorre scrivere un *ciclo PLC*, cioé un programma, che viene eseguito da EZ-Red. Durante l'esecuzione del programma il modulo EZ-Red può rimanere autonomo, ma è anche possibile usare il computer per interagire con il PLC. Quando EZ-Red esegue un ciclo PLC, ci possono essere tre scenari:

1. Esecuzione completamente autonoma. E' l'ideale nei casi dove non sono richieste funzioni avanzate come un'interfaccia utente grafica, accesso a memorie di massa, o calcoli complessi. Tutta la periferia può essere gestito direttamente da EZ-Red+ciclo PLC: è il caso di semplici automazioni di processo.

2. Esecuzione combinata EZ-Red + Computer; il computer esegue prevalentemente letture. EZ-Red svolge il ciclo, mentre il computer interroga ciclicamente il modulo, allo scopo di visualizzare l'andamento del ciclo, o acquisire e registrare dati.

3. Esecuzione combinata EZ-Red + Computer, con piena interazione. EZ-Red svolge il ciclo o alcune parti di esso (probabilmente le parti con temporizzazione più critica), mentre il computer può influire sull'andamento del ciclo: avvio/arresto, modifica di tempi, scelta di cicli diversi e così via. In questa configurazione è possibile che il computer faccia uso della sua potenza e versatilità, come l'uso di database, connessioni di rete, calcoli di formule complesse. EZ-Red funziona come una subroutine o un task parallelo all'interno di un sistema più articolato.

Compilatore TSmon

Il programma PLC deve essere preparato con un editor di testi o usando l'editor interno di TSMON.EXE, l'applicativo fornito insieme a EZ-Red, che è un vero e proprio ambiente integrato (IDE). Se si usa un editor esterno, esso deve generare file di testo semplici: i word processor non sono adatti. Il programma TSMON permette di creare e modificare il ciclo; inviarlo a EZ-Red, metterlo a punto e monitorarlo. Quando il ciclo risultante è corretto, va memorizzato (store) nella memoria non volatile (flash) del modulo, altrimenti all'accensione successiva EZ-Red carica il ciclo precedente (l'ultimo che era stato salvato nella memoria flash).

Riferirsi al manuale d'uso di TSmon per maggiori informazioni sul programma.

Linguaggio EZ-Red

Introduzione

Il programma del ciclo PLC è in grado di leggere gli ingressi, impostare le uscite, memorizzare valori intermedi, eseguire calcoli, dirigere lo svolgimento delle operazioni (controllo di flusso); si tratta perciò di un vero e proprio linguaggio di programmazione, simile in molti aspetti al BASIC, soprattutto per il fatto che è *procedurale* (le operazioni sono eseguite in sequenza come sono scritte) e *imperativo* (le operazioni sono descritte da verbi imperativi come GOTO, SUSPEND, WAIT. Ingressi e uscite del modulo vengono assimilati a variabili che possono essere lette e scritte: un comando come "Y1=ON" provoca l'attivazione dell'uscita digitale 1, mentre un comando come "Y1=X1" assegna all'uscita 1 lo stesso valore presente all'ingresso 1 - in un solo comando vi sono la lettura di un ingresso e l'impostazione di un'uscita.

Un comune linguaggio per computer, tuttavia, non è molto adatto a un modulo come EZ-Red, che è molto vicino a un PLC e quindi, come tale, ha bisogno di comandi adatti a gestire gli I/O e i tempi precisi che un ciclo macchina richiede. Per questo motivo il linguaggio EZ-Red è un ibrido che permette di usare sia il modello procedurale (come il BASIC) sia il modello funzionale (come il Ladder usato sui PLC), e anche di usarli contemporaneamente nello stesso ciclo. Dal linguaggio Ladder eredita il concetto delle *risorse*: queste hanno un nome preciso, per esempio Y1 (la prima uscita di potenza), AOUT2 (la seconda uscita analogica), oppure TIMERMS1 (un timer che conta millisecondi). Queste risorse vengono manipolate come se fossero le variabili di un linguaggio di programmazione comune, ma con estensioni tipiche di un PLC, per esempio il concetto di fronte di cambiamento di un segnale (fronti di salita e discesa), che vengono espressi in modo simile al Ladder ma con una sintassi più classica.

Tipi di dato e Risorse

Quasi tutti i linguaggi di programmazione hanno il concetto di tipo di dato (alcuni permettono anche di definire nuovi tipi). EZ-Red possiede solo tre tipi: numeri interi di 16 bit (da 0 a 65535), il sottotipo byte da 8 bit, e i valori booleani o BIT (di un bit), che hanno solo il valore on/off, acceso/spento, o vero/falso. Il tipo di dato BIT si usa per ingressi e uscite digitali, che possono avere solo due stati. I numeri interi si usano per fare calcoli, e possono essere legati agli I/O analogici e alla misura del tempo (in millisecondi, secondi, minuti).

Le Risorse sono tutti gli ingressi, le uscite, e le memorie interne - in poche parole, tutto l'hardware di EZ-Red che può essere manipolato dal PLC. Ogni risorsa è di un tipo preciso: per esempio Y1 (uscita di potenza 1) è di tipo BIT, mentre AIN1 (ingresso analogico 1) è di tipo INTERO. Le risorse vengono lette specificandone il nome in un'*espressione*, e vengono impostate usando un'assegnazione. Così, l'istruzione Y1=X1 provoca la lettura di X1 (ingresso 1, di tipo BIT), e assegna il valore a Y1 (uscita 1, anch'essa di tipo BIT). La parte a sinistra del segno di uguale deve essere un nome di risorsa; la parte a destra del segno uguale deve essere un'*espressione* di tipo compatibile. In "Y1=X1" l'espressione X1 è di tipo BIT, e quindi compatibile con Y1 - anch'essa di tipo bit. Le principali risorse, oltre a ingressi e uscite, comprendono i dati "R" (relé interni virtuali, variabili di tipo BIT) e "DT", variabili interne di tipo INTERO; gli identificatori R e DT servono per mantenere dati intermedi ed effettuare calcoli. Vi sono poi altre risorse utili per leggere o impostare stati interni e funzioni particolari di EZ-Red.

Riferirsi al paragrafo Espressioni per maggiori informazioni.

DEFINE e DECLARE

La parola chiave DEFINE serve per assegnare un nome significativo a una risorsa o una costante, in modo da rendere più comprensibile il testo del ciclo. Usando per esempio "DEFINE MOTOR Y1", è possibile riferirsi all'uscita Y1 con il nome della funzione di tale uscita (l'avviamento del motore, in questo caso); nel testo successivo del ciclo è quindi possibile scrivere "MOTOR=ON" e "MOTOR=OFF" per accendere e spegnere il motore collegato all'uscita 1 (tra l'altro, anche ON e OFF sono due identificatori dichiarati automaticamente da TSMON con *DEFINE*). DEFINE è simile al #define del linguaggio C, ma più restrittivo.

Se la risorsa usata nel DEFINE è un ingresso o una uscita (Xn o Yn), è possibile indicarne una eventuale negazione usando un punto esclamativo. Per esempio:

```
DEFINE stop !X2
```

crea un identificatore "STOP" che si riferisce a X2, ma negato. Quando l'ingresso digitale 2 ha tensione, STOP contiene OFF (**e anche X2**), e viceversa. Questa possibilità può servire per "raddrizzare" la logica di ingressi o uscite che abbiano una logica negata. Per esempio, un pulsante di arresto ciclo di solito è normalmente chiuso e, collegato all'ingresso X2, fornirebbe "ON" in condizioni di riposo. Ne consegue che sarebbe poco appropriato nominarlo STOP, poiché in condizioni normali (pulsante non premuto) risulterebbe che "STOP" è "ON". La negazione durante un DEFINE agisce su un'impostazione (o risorsa) di EZ-Red, XINVERT e YINVERT, che servono appunto a negare ingressi e uscite.

La parola chiave DECLARE è simile a DEFINE, ma serve a creare (dichiarare) variabili, non ad associare nomi a risorse preesistenti o costanti, e richiede anche un segno di uguale seguito da un'espressione. L'effetto è esattamente quello di creare una variabile, di tipo bit o intero, e assegnarle un valore. La sintassi completa è:

```
DECLARE [LOG] [DT|R] identificatore = espressione
```

I modificatori "R" o "DT" non sono normalmente necessari: il tipo di dato (bit o integer) assegnato all'identificatore viene ricavato dal tipo di espressione; solo nel caso di una dichiarazione "auto-ricorsiva", come:

```
DECLARE R MARCIA = MARCIA OR START
```

il modificatore "R" è necessario. Serve perché l'espressione a destra dell'uguale contiene il termine che si sta definendo e che perciò è ancora sconosciuto, e quindi non se ne conosce il tipo.

Il modificatore "LOG" indica che le modifiche (assegnazioni) alla variabile appena creata causeranno un "evento": una comunicazione asincrona al PC collegato ad EZ-Red (vedere il comando LOG per ulteriori informazioni).

Che si usi DEFINE o DECLARE, l'identificatore che si sta dichiarando deve essere nuovo: non è possibile ridefinire identificatori già esistenti. Per DEFINE, inoltre, la risorsa da associare al nuovo identificatore deve essere già esistente. In entrambi i casi, un commento sulla stessa riga, o su una riga precedente contenente solo il commento, associano tale commento all'identificatore:

```
define motore y3    ; questo commento diventa la descrizione di MOTORE

; questo commento viene associato a contacicli
declare contacicli = 0
```

Nel programma TSMON (compilatore e ambiente di sviluppo di EZ-Red), premendo il tasto F1 su un identificatore la descrizione associata viene visualizzata nell'area dei suggerimenti:

Multitask

Il modello procedurale, rispetto al ladder, è generalmente più comprensibile perché è possibile concentrare l'attenzione sulle singole istruzioni, eseguite una dopo l'altra. Questo vantaggio però comporta anche l'esatto svantaggio contrario: è difficile con un linguaggio procedurale eseguire due compiti simultanei. Per esempio, è facile far lampeggiare un'uscita alla frequenza di 1 hertz:

```
ripeti:
  y1=on
  wait 500          ; <--- attesa di 500 ms
  y1=off
```

```
        wait 500          ; <--- attesa di 500 ms
        goto ripeti
```

ma diventa molto più complicato far lampeggiare due uscite a due frequenze diverse. Il problema si può risolvere senza complicazioni prevedendo più *Task* (compiti), ognuno dei quali funziona in modo indipendente dall'altro. Per far lampeggiare due uscite a due frequenze diverse basta usare due programmi identici a quello mostrato qui sopra; ognuno dei due Task fa lampeggiare un'uscita, in modo semplice e indipendente, senza usare timer o altro hardware, reale o virtuale:

```
Task1:
  y1=on
  wait 500
  y1=off
  wait 500

Task2:
  y2=on
  wait 300
  y2=off
  wait 300
```

Il contenuto del task 1 termina in corrispondenza dell'inizio del task 2. Per definizione i task sono ripetitivi: quando non ci sono più istruzioni da eseguire, il task riparte dalla sua prima istruzione. Quello che fa il frammento di codice è semplice: il Task 1 accende l'uscita 1, attende 500 millisecondi, poi la spegne; attende ancora 500 millisecondi e, implicitamente, ricomincia. Il risultato è un'onda quadra con periodo di 1 secondo. Il Task 2 è uguale, a parte il tempo di attesa. Si può notare che il linguaggio è procedurale, e sarebbe perciò impossibile generare due onde quadre usando un solo task.
Con EZ-Red è possibile definire fino a 16 Task contemporanei, ed è perciò possibile avere alcuni vantaggi della programmazione di tipo ladder senza abbracciarne fino in fondo la filosofia funzionale. I task, inoltre, possono essere sospesi e riavviati.

I/O: sincrono (default) o asincrono

I classici PLC aggiornano gli I/O una volta per ciclo (in modo *atomico*; dispongono di una copia in memoria degli ingressi e delle uscite): così facendo, semplificano la logica di programmazione, ma nello stesso tempo introducono limiti che in in alcuni casi diventano troppo vincolanti. L'aggiornamento "*una volta per ciclo*" sarebbe problematico con EZ-Red, per via del suo approccio procedurale e della presenza di task concorrenti. Quindi, normalmente, ingressi e uscite sono aggiornati automaticamente ogni volta che il Task1 salta "all'indietro" (ciò comprende il caso del riavvio implicito del task). Inoltre una istruzione WAIT, se eseguita nel Task1, aggiorna per prima cosa le uscite; per gli ingressi non è necessario. Questo meccanismo di aggiornamento comporta che i task diversi dall'1 debbano usare una sintassi particolare se devono aggiornare una singola uscita, oppure possono forzare un aggiornamento. Il Task2 dell'esempio soprastante in effetti dovrebbe usare, per funzionare correttamente, la sintassi speciale come segue:

```
Task2:
  y2<=on
  wait 300
  y2<=off
  wait 300
```

Come si vede, si usa un "<=" invece di un semplice "=". Questa sintassi comporta l'aggiornamento immediato dell'uscita specificata. Un'alternativa è usare i comandi UPDATEX/UPDATEY/UPDATEXY.

Un'altra possibilità è quella di impostare AUTOUPDATEXY a TRUE; così facendo, ingressi e uscite vengono aggiornati dopo ogni passo di programma, di qualsiasi task.
Se AUTOUPDATEXY è impostato (normalmente è OFF), un potenziale problema si può verificare con sequenze simili alla seguente:

```
; muovere il motore avanti o indietro
mot_avanti   = avanti
mot_indietro = NOT avanti
```

In questo spezzone due uscite, *mot_avanti* e *mot_indietro*, comandano la marcia di un motore nelle rispettive direzioni e non è desiderabile che entrambe le uscite siano contemporaneamente ON. Con AUTOUPDATEXY a ON, però, può succedere: supponendo che da un istante precedente *avanti* sia OFF, per cui l'uscita *mot_indietro* dovrebbe essere ON, quando avanti diventa ON anche *mot_avanti* lo diventa - prima che *mot_indietro* diventi OFF. Il problema si può risolvere facilmente circondando le istruzioni critiche con due altre istruzioni:

```
; muovere il motore avanti o indietro
AUTOUPDATEXY = FALSE      ; disabilita temporaneamente
mot_avanti   = avanti
mot_indietro = NOT avanti
AUTOUPDATEXY = TRUE       ; riabilita
```

Il frammento esposto dovrebbe risultare familiare a chi conosce gli *interrupt*: l'I/O asincrono viene inibito per un breve tempo, e poi riabilitato; questo assicura che un aggiornamento non possa avere luogo in un momento indesiderato.

Secondo il tipo di ciclo che s'intende implementare, vi sono due strategie di base.
a) Lasciare AUTOUPDATEXY a OFF (valore di default) e usare Task1 come ciclo continuo senza WAIT, alla stregua di un PLC. Gli altri task possono basarsi sull'aggiornamento automatico eseguito da Task1, se veloce a sufficienza, oppure impostare direttamente le uscite con la sintassi "Y1<=ON" e simili. Le istruzioni WAIT leggono gli ingressi basandosi sempre sui dati reali e non sulla copia in memoria.

b) Impostare AUTOUPDATEXY a ON e tenere presente che tutti gli ingressi e le uscite sono sempre aggiornati, praticamente senza alcuna copia in memoria. Eventuali sezioni di codice critico, come quello mostrato sopra, possono essere protette come mostrato.

Attenzione: la lettura delle risorse Y ritorna sempre l'ultimo valore impostato, anche se l'uscita fisica di tensione non è stata ancora aggiornata. Il seguente frammento di codice non ha alcun effetto sulla tensione dell'uscita 1 (Y1):

```
autoupdateXY = false
Y1 = on             ; Y1 è ON, ma la tensione d'uscita non è ancora aggiornata
Y1 = not Y1         ; la lettura di Y1 rispecchia già lo stato ON, quindi diventa OFF
```

Struttura del programma

Il programma (o ciclo PLC) è formato da una serie di istruzioni che fungono come INIT (predisposizione), seguita da una serie di Task. Ogni task è contrassegnato da una etichetta TASK**x**, con **x** che va da 1 a 16, e termina con l'inizio del task successivo o con la fine del programma. Sia l'INIT sia i task sono opzionali. Se si dichiarano Task, essi devono cominciare con il numero 1 e devono essere consecutivi. Se esiste una parte di codice prima di Task1, tale parte di codice viene eseguita una volta sola, come iniziazione.

```
; parte di INIT (opzionale), che precede qualsiasi "TaskX:"
y1=x1        ; eseguita una volta sola all'avvio
wait 1000    ; eseguita una volta sola all'avvio

Task1:       ; inizio del primo task
  ...        ; istruzioni eseguite ciclicamente
  ; è inserito implicitamente un GOTO Task1
```

Se si omette qualsiasi etichetta "TaskX:", il programma consiste della sola parte di INIT e verrà eseguito una volta sola. Se la prima istruzione utile è una etichetta del tipo "TaskX:", allora la parte di INIT è assente e l'esecuzione parte da Task1.

```
wakeup 2     ; questa istruzione è parte dell'init

Task1:
  y1=x1
  wait 1000
  ; istruzione GOTO implicita che fa ripartire Task1
  ; questo task copia continuamente X1 su Y1, "a strattoni"

Task2:
  y2=seconds and 1<>0     ; "seconds" conta i secondi
  ; qui un altro GOTO implicito fa ripartire Task2
  ; questo task fa lampeggiare y2 ogni due secondi
```

Il programma qui sopra mostra due task. Si noti l'istruzione "wakeup 2", che serve ad attivare il task 2: all'avviamento, tutti i task a parte il primo (Task1) sono sospesi.

Formato del programma

Un programma EZ-Red è formato da una serie di righe di testo; le righe vuote vengono ignorate, e si possono usare per aumentare la leggibilità del programma. La differenza fra lettere maiuscole e minuscole è sempre ignorata: il testo si può scrivere indifferentemente nei due modi.

Commenti

Il programma può contenere commenti, introdotti da un punto e virgola, che durano fino alla fine della linea, oppure commenti multilinea che iniziano con una parentesi quadra aperta e terminano con una chiusa:

```
; questo è un commento. La riga successiva è ignorata perché vuota

y1=x1        ; questo è un altro commento
[
questo è un commento
composto da più linee
]
```

Identificatori

Gli identificatori sono parole inventate dal programmatore, per dare un nome a ingressi, uscite, variabili, etichette eccetera. Devono iniziare con una lettera o un segno di sottolineatura, e proseguire con una lettere, sottolineature, o cifre numeriche. I numeri devo essere composti solo da cifre numeriche.

Non è possibile definire (dichiarare) due volte lo stesso identificatore; molti identificatori (tutte le risorse interne) sono già definite dal compilatore e non possono quindi essere ridefinite.

Numeri (letterali numerici)

Il linguaggio prevede solo interi senza segno, quindi le costanti letterali ("12", "0", "16384") devono essere formati solo da cifre, senza punti o altri segni. Però è possibile usare costanti in formato binario o esadecimale, rispettivamente usando la notazione "0b" (zero-bi) o "0x" (zero-ics, tipica del linguaggio C):

```
ylampmask1 = 0b1011       ; decimale 11
bcw_chr = 0x55            ; binario 0101.0101
```

Etichette (dichiarazione)

Una riga può cominciare con un'etichetta, che è un identificatore seguito da due punti. Dato che gli identificatori non possono essere ridefiniti, ne consegue che non possono esistere due etichette uguali. La riga contenente l'etichetta può proseguire con una istruzione o un commento, o entrambi, ma generalmente si lascia solo l'etichetta per aumentare la leggibilità.

Le etichette nella forma "TaskX", dove X è un numero, sono speciali perché definiscono l'inizio di un task. Come accennato, il primo task definito deve chiamarsi Task1, e ogni nuovo task dichiarato deve essere numericamente consecutivo al precedente. Il numero massimo di task è 16.

All'interno di un blocco IF/THEN/ELSE non è possibile dichiarare etichette, anche se è possibile farci riferimento (con GOTO).

DEFINE (dichiarazione)

La dichiarazione DEFINE permette di assegnare un nome a una risorsa (ingresso, uscita, memoria interna...), oppure assegnare un nome a una costante numerica:

```
DEFINE motore       y2     ; assegna un nome (e questa descrizione) a Y2
DEFINE fine_corsa   !x1    ; si possono negare ingressi e uscite digitali
DEFINE tempo_attesa 2000   ; definisce una costante
```

Se si usa la negazione con "!" (punto esclamativo) in una DEFINE, il compilatore inserisce un'istruzione invisibile nella parte INIT; ciò ha l'effetto di negare l'identificatore che si sta definendo, **e anche l'ingresso o l'uscita relativa**.

Un commento dopo l'istruzione DEFINE associa una descrizione al nome simbolico.

DECLARE (dichiarazione)

La dichiarazione DECLARE crea una variabile, di tipo bit o integer, e vi assegna un valore:

```
; DECLARE [LOG] [R|DT] id = espressione
declare tempo = 20              ; crea un integer, con il valore 20
declare log x1oppurex2 = x1 or x2
  ; crea un bit e assegna (solo all'esecuzione) l'OR di X1 e X2
  ; ogni variazione di x1oppurex2 sarà inviata (notificata) al PC

define avvio x1

; questo commento viene associato alla dichiarazione qui sotto di “marcia”
declare r marcia = marcia or avvio
declare dt cicli_eseguiti = cicli_eseguiti+1
  ; se l'espressione contiene l'identificatore da definire, occorre
  ; qualificare la DECLARE con "R" oppure "DT"
```

Il modificatore LOG è valido solo per le dichiarazioni di tipo BIT.

Un commento dopo l'istruzione DECLARE associa una descrizione al nome simbolico.

Istruzioni (statement)

Le istruzioni differiscono dalle dichiarazioni perché generano effettivamente codice per il PLC di EZ-Red (le *DEFINE* con negazione sono un'eccezione). Vi sono tre tipi di istruzioni: *assegnazioni*, *comandi*, e comandi di *controllo di flusso*. Quasi sempre si usano *espressioni*, che sono una combinazione di identificatori (*risorse*), costanti letterali, e *operatori*.

Assegnazioni

Un'assegnazione ha la forma "risorsa = espressione". Praticamente ogni operazione che agisce sull'hardware viene eseguita tramite un'assegnazione: per attivare un'uscita di potenza si scrive per esempio "Y1 = ON", e per portare l'uscita analogica 2 a 10 volt si scrive "AOUT2 = 255".

La parte a destra del segno uguale dev'essere un'espressione, che può essere una semplice costante, il contenuto di una variabile (memoria interna), o una combinazione di molti termini. Le espressioni sono spiegate in dettaglio nel paragrafo seguente. Il tipo di dato (cioé, bit o intero) fornito dall'espressione deve corrispondere al tipo di dato a sinistra del segno uguale.

Nel caso di assegnazione delle uscite digitali Y1..Y8, lo stato reale dell'uscita viene aggiornato effettivamente solo quando il Task1 esegue un salto all'indietro o in occasione di una istruzione WAIT. Però, lo stato “interno” delle uscite (la copia in memoria) è aggiornata immediatamente. Per provocare l'aggiornamento immediato di una uscita (questo è utile nei task diversi da Task1), usare la seguente sintassi:

```
Y2 <= ON            ; aggiornamento sempre immediato
```

Riferirsi al capitolo “I/O sincrono o asincrono” per maggiori informazioni.

Espressioni

Un'espressione è una serie di valori combinati tramite operatori. I valori possono essere specificati da costanti letterali, variabili, o identificatori di risorse. Un'espressione applica i calcoli definiti dai vari operatori e fornisce un unico risultato. Sono espressioni, per esempio, "1+5", “cicli+1”, "x1 OR x2", “x1 or ain1>ain2+4”.

Le espressioni devono iniziare con un “valore”, cioè un identificatore o un letterale, e proseguire con coppie di "operatore valore". Al posto di un “valore” si può usare un'altra espressione, racchiusa fra parentesi. Esempi di espressioni sono:

```
3                   ; l'espressione più semplice, un singolo letterale
x1                  ; un singolo termine (“valore”)
x1 or x2            ; due termini combinati con OR
temp+sonda/2        ; tre termini (e quindi due operatori)
(temp+sonda)/2      ; due termini, il primo dei quali è a sua volta un'espressione
```

Ogni operatore accetta solo un certo tipo di operandi (termini); per esempio, l'operatore “+” accetta solo operandi di tipo integer; però alcuni operatori come AND e OR esistono nella doppia versione – *logica*, che opera sui bit (termini booleani), e *bitwise*, che opera su valori integer. Di solito gli operatori accettano dati di un tipo e forniscono come risultato lo stesso tipo; alcuni operatori, però, accettano operandi integer e restituiscano boolean (bit), come l'operatore “<”, che confronta due integer e fornisce la risposta come bit. Infine, diversi operatori hanno diverse precedenze. L'espressione aritmetica “12+4 / 2” fornisce il risultato 14 (non 8!), perché l'operatore di divisione “/” ha precedenza maggiore dell'operatore di somma “+”. In ordine di precedenza crescente, gli operatori di EZ-Red sono:

OR logico	XOR logico	= (bit)	<> (bit)			dati bit / booleani operazioni logiche
AND logico						
<	>	<=	>=	= (int)	<> (int)	dati int, risultato bit
+ (più)	- (meno)					operatori aritmetici
* (molt.)	/ (divis.)					
OR (int)	XOR (int)					operatori bitwise
AND (int)						

EZ-Red supporta una sintassi alternativa: invece di "x1 or x2 or x3" è possibile usare "or x1 x2 x3".

Operatori

Per i tipi bit (booleani o logici) sono previsti i seguenti operatori:

Operatore	Operandi	Risultato	Spiegazione
AND	bit, bit	bit	AND logico. Il risultato è TRUE se entrambi gli operandi sono TRUE.
OR	bit, bit	bit	OR logico. Il risultato è TRUE se almeno uno dei due operandi è TRUE.
XOR, <>	bit, bit	bit	XOR logico. Il risultato è TRUE se i due operandi sono diversi.
=	bit, bit	bit	Uguaglianza logica. Il risultato è TRUE se i due operandi sono uguali.

Per i tipi numero sono previsti i seguenti operatori:

Operatore	Operandi	Risultato	Spiegazione
AND	num, num	numero	AND aritmetico. Ogni bit del primo operando viene messo in AND con il bit corrispondente del secondo operando. Il risultato contiene i 16 bit elaborati.
OR	num, num	numero	OR aritmetico.
XOR	num, num	numero	XOR aritmetico.
=	num, num	bit	Uguaglianza numerica. Il risultato è TRUE se i due operandi sono uguali.
<>	num, num	bit	Disuguaglianza numerica (simile a XOR). Il risultato è TRUE se i due numeri sono diversi.
<	num, num	bit	Confronto numerico. TRUE se il primo operando è inferiore al secondo.
<=	num, num	bit	Confronto numerico. TRUE se il primo operando è inferiore o uguale al secondo.
>	num, num	bit	Confronto numerico. TRUE se il primo operando è maggiore del secondo.
>=	num, num	bit	Confronto numerico. TRUE se il primo operando è maggiore o uguale al secondo.
+, -	num, num	numero	Somma / sottrazione aritmetica.
***, /**	num, num	numero	Moltiplicazione / divisione (intera) aritmetica

NOT (operatore / funzione)

Gli operatori visti finora vogliono due argomenti. L'operatore (o meglio dire funzione) NOT è speciale perché nega l'intera espressione alla sua destra. Se l'espressione è di tipo booleano (bit), viene eseguita una negazione

logica; se l'espressione è numerica (integer) avviene una negazione bitwise, in complemento a 2.

Operatori (modificatori) speciali di bit: /, \, ^, !

Nel campo d'applicazione tipico di EZ-Red è facile dover gestire negazioni e fronti di segnali logici. Per facilitare la gestione dei fronti sono disponibili alcuni modificatori da apporre immediatamente a sinistra dell'identificatore:

Modificatore	Significato
/	Fronte di salita (true solo se il bit è ON mentre prima era OFF)
\	Fronte di discesa
^	Qualsiasi fronte (true solo se il bit è cambiato)

L'espressione "/X1" risulta vera se, quando valutata, X1 è ON ma nella valutazione precedente era OFF.

Si consideri il seguente caso: si vuole accendere l'uscita Y1 quando il pulsante collegato a X1 viene premuto – e solo in quell'istante; se X1 continua a essere premuto, l'uscita Y1 non deve cambiare più:

```
if /X1 then Y1=on
```

A ogni valutazione di "/X1", viene fatto un confronto con il valore precedente di X1 e il nuovo stato - di X1, non del risultato dell'espressione - viene memorizzato internamente. Nello spezzone di codice seguente:

```
if /X1 then Y1=on
if /X1 then timersec2 = 15
```

sono coinvolti due "stati precedenti" (perché ci sono due espressioni); benché si riferiscano entrambi a X1, sono distinti. Questo è diverso da quello che fanno normalmente i PLC.

Si noti anche che scrivendo:

```
Y1=/X1
```

si ottiene probabilmente che l'uscita Y1 sia ON per tempi brevissimi, o meglio, per il tempo che intercorre tra la valutazione di questa assegnazione e la successiva esecuzione dello stesso punto del programma, nel "giro" successivo (di solito i Task sono ciclici).

L'uso tipico di questa notazione per i fronti si ha nei costrutti IF-THEN:

```
if /pulsante then timersec1=10
```

oppure in espressioni di auto-ritenuta o inversione di stato tipica dei PLC, come:

```
marcia = marcia xor /avvio
```

che inverte lo stato di "marcia" a ogni pressione del pulsante "avvio".

Istruzione di attesa WAIT

WAIT è un'istruzione complessa che serve per attendere zero o più eventi (fino a 4), con un tempo massimo di attesa. La forma senza eventi è:

WAIT [timeout_in_millisecondi]

e comporta una sospensione temporanea del task per il tempo specificato. Il tempo di timeout può essere specificato con una costante o un nome di risorsa ma non un'espressione (però si può calcolare un'espressione in una riga precedente). Se il timeout è zero oppure omesso, il task viene sospeso analogamente all'istruzione SUSPEND.

Che sia specificato un timeout o no, è possibile indicare fino a quattro eventi da attendere. Un evento è uno stato di un bit: occorre specificare il bit, opzionalmente preceduto dal modificatore di negazione "!". L'istruzione WAIT attende che uno qualsiasi dei quattro eventi si presenti (un OR logico):

```
WAIT                ; sospende il task corrente
WAIT   0            ; sospende il task corrente
WAIT 1000           ; attende un secondo

WAIT X1 X2          ; attende che X1 o X2 siano ON
WAIT !TMS1          ; attende che il timer T1 cada

WAIT 100 X1         ; attende X1 on, per 100 ms

WAIT 1000 X1 X2 !TMS1 !Y2
  ; attende, al massimo un secondo, che X1 o X2
  ; diventino ON oppure che TMS1 o Y2 siano OFF

WAIT start          ; attende start=on, per sempre
```

Quando l'istruzione WAIT specifica eventi, se il timeout è omesso oppure zero, il tempo d'attesa è infinito.

Precisazione sul tempo d'attesa: se si specifica ***n*** come timeout, il tempo effettivo va da ***n***-1 a ***n*** millisecondi. Per esempio, "WAIT 1" attende *al massimo* 1 millisecondo. Per attendere *almeno* 1 millisecondo, occorre specificare "WAIT 2". Il motivo di questo comportamento è il seguente. Supponendo di volere generare un'onda quadra con periodo di 2 millisecondi, il seguente codice non funzionerebbe:

```
Task4:
  ; genera un'onda quadra del periodo di 2 ms
  y2 <= on
  wait 1
  y2 <= off
  wait 1
```

Il motivo è che le istruzioni come "y2=on", "y2=off" richiedono tempo per essere eseguite, e questo tempo si somma ai due millisecondi specificati dalle due WAIT. L'onda quadra non avrebbe un periodo di 2 millisecondi, ma sicuramente maggiore, anche se di poco. Con l'implementazione di EZ-Red, invece, l'onda quadra generata è esattamente di 500 hz (periodo=2 ms). Questo accade perché l'istruzione WAIT è sincronizzata su un tick interno di 1 Khz; l'effetto è quello di ritardare al massimo *quasi* 1 millisecondo in meno di quanto specificato, ma il tempo risparmiato è quello necessario a eseguire le altre istruzioni.

WAITREMAIN (tempo residuo dopo WAIT)

Quando l'istruzione WAIT termina, il tempo residuo di attesa viene scritto in WAITREMAINx, dove x è il numero del task che ha eseguito il WAIT. Questo tempo risulta zero se l'attesa è terminata perché il timeout è scaduto (o non era stato specificato), ed è diverso da zero se l'istruzione è terminata a causa del verificarsi di uno degli eventi attesi. L'uso di questa risorsa può servire per eseguire più WAIT in cascata, con un tempo totale:

```
[ Task1 ]
  y1 = on
  wait 1000 ...
  wait waitremain1 ...
  y1=off
```

Nel frammento qui sopra, Y1 rimane accesa per non più di 1 secondo, indipendentemente dagli eventi specificati nelle istruzioni WAIT.

Risorse principali (ingressi, uscite, hardware in genere)

“Risorse” è un termine generico per indicare parti hardware, come ingressi e uscite, ma comprende anche dispositivi virtuali, implementati via software e flag, opzioni, variabili e via dicendo. Perfino le costanti ON e OFF sono in realtà nomi di risorse. Quando una risorsa è singola, un nome abbastanza descrittivo viene usato per essa, come FEEDBACKS o WDTOUTS; quando vi sono più risorse identiche, come gli ingressi digitali che sono 8, viene usato un nome, come "X", seguito da un numero progressivo. Per esempio gli ingressi digitali, che sono 8, si chiamano X1, X2..., X8. Esiste anche però una risorsa di un byte, XBYTE, che contiene lo stato di tutti gli ingressi X insieme, un bit per ogni ingresso.

La tabella seguente mostra un elenco delle principali risorse disponibili. La colonna Quantità riporta il numero di risorse disponibili con quel nome. Per esempio, gli ingressi digitali sono X1..X8, per cui il nome è "X", e la quantità è 8. Più avanti in questo documento verranno descritte più risorse e con maggior dettaglio.

Nome	Quantità	Spiegazione
TRUE, ON	-	Costante che indica accensione, livello logico alto: es. Y1=TRUE; Y2=ON.
FALSE, OFF	-	Costante che indica spento, livello basso: es. Y1=False; Y2=Off
X	8	Uno degli 8 ingressi digitali, da X1 a X8. Lo stato degli ingressi è filtrato e e può essere invertito: vedere il paragrafo specifico.
XINVERT	8	Imposta l'inversione logica del relativo ingresso digitale X.
XCOUNT	2	Conteggio dei fronti di salita sugli ingressi X1 e X2, da 0 a 65535.
FX	2	Stato ON/OFF degli ingressi veloci. Anche se questi ingressi sono progettati per il collegamento di un encoder, possono essere letti come tutti gli altri. Non dispongono però di filtraggio e inversione.
FXCOUNTL FXCOUNTH	2	Contano i fronti di salita degli ingressi FX. Il valore numerico di FXCOUNTL va da 0 a 65535 (16 bit), poi si azzera ed FXCOUNTH si incrementa di 1.
Y	8	Una delle otto uscite di potenza, da Y1 a Y8. Impostandola a ON (o TRUE), l'uscita presenta tensione (c'è però un'opzione per invertire le uscite).
YINVERT	8	Imposta l'inversione logica della relativa uscita di potenza Y.
YLAMPMASK	8	Risorsa per far lampeggiare regolarmente, con una maschera, le uscite.
TIMERMS	6	6 timer a 16 bit che contano millisecondi (ms), TIMERMS1..TIMERMS6. Vedere il paragrafo apposito per i timer.
TIMERSEC	6	6 timer a 16 bit che contano secondi. Vedere il paragrafo relativo.
TIMERMIN	4	4 timer a 16 bit che contano minuti. Vedere il paragrafo relativo.
TMS	6	Stato On/Off del relativo timer TIMERMSn.
TSEC	6	Stato On/Off del relativo timer TIMERSECn.
TMIN	4	Stato On/Off del relativo timer TIMERMINn.
MILLISECS	-	Contatore libero, a 16 bit (raggiunto 65535 si azzera) che conta millisecondi
CENTISECS	-	Contatore libero, a 16 bit, che conta centesimi di secondo
SECONDS	-	Contatore libero, a 16 bit, che conta secondi
R	64	Relé interni, a bit, da usare come memorie generiche.
DT	64	Memorie interne, numeriche a 16 bit. Contengono numeri da 0 a 65535.
AOUT	2	Uscite analogiche 0-10V. I numeri utilizzabili vanno da 0 (0 volt) a 255 (10 volt).
AIN	2	Ingressi analogici. Ritornano il valore di tensione presente all'ingresso come numero compreso tra 0 (0 volt) e 255 (10 volt).
ENCODERL, ENCODERH	-	Posizione dell'encoder opzionale collegato agli ingressi veloci FX1 ed FX2. Fare riferimento ala paragrafo relativo.

Ingressi digitali X1..X8

Le risorse X1..X8 rispecchiano lo stato degli ingressi digitali da 1 a 8, e dispongono di alcune opzioni per la loro gestione. La prima di queste è l'*inversione logica* dell'ingresso. Se si pensa a un fine corsa inteso a limitare o terminare la corsa di un asse, è logico immaginare di utilizzare un contatto normalmente chiuso, che porta 24 volt su un ingresso. Quando l'asse arriva sul fine corsa, il contatto si apre e la tensione sull'ingresso scende a zero. Nel programma si potrebbe definire "DEFINE FINECORSA X1". In questo caso, la presenza logica del fine corsa è indicata dalla mancanza di tensione, quindi dal valore OFF. In tutto il ciclo occorrerebbe ragionare in logica negativa: per esempio per accendere l'uscita 1 quando l'asse è sul fine corsa bisognerebbe scrivere "Y1 = FINECORSA <> ON" (o "Y1=!finecorsa") che significa: accendere l'uscita 1 se FINECORSA è diverso da ON. Naturalmente sarebbe meglio scrivere "Y1 = FINECORSA", ragionando in logica positiva. Per invertire la logica di un ingresso si può usare un DEFINE:

```
DEFINE finecorsa !x1
```

XINVERT (inversione logica degli ingressi)

L'effetto della DEFINE esposta è semplicemente quello di alzare XINVERT1 nella sezione INIT nascosta del ciclo PLC. Alternativamente, e in qualsiasi momento, è possibile modificare XINVERT1..XINVERT8.

Si noti che la notazione "!" per invertire una risorsa in un'espressione è un meccanismo separato. Usando la DEFINE sopra, e riferendosi a "!finecorsa", in realtà si usa X1 senza inversioni.

XTHRESHOLD UP/DN (filtro antidisturbo / antirimbalzo)

Il trasferimento del livello di tensione dall'ingresso alle risorse X dipende anche da un filtro software volto a eliminare disturbi e rimbalzi. Normalmente non c'è alcun filtro, e gli ingressi vengono *campionati* (controllati) 1000 volte al secondo. Una variazione da 0 a 24 volt viene perciò percepita in un millesimo di secondo. A volte questo non è desiderabile, magari a causa di possibili disturbi, o a causa di rimbalzi dei contatti esterni (switch, pulsanti e simili). Il gruppo di variabili XTHRESHOLDUP (ce ne sono 8) permette d'impostare, in millisecondi, l'attesa della presenza stabile della tensione, prima di portare l'ingresso logico X a ON. Se la soglia è messa a 5 millisecondi, per esempio, disturbi di 3 o 4 millisecondi sul segnale vengono cancellati. La soglia comporta però un ritardo di risposta, che è il tempo di attesa necessario a verificare che il segnale sia stabile.

Analogamente a XTHRESHOLDUP, XTHRESHOLDDN imposta il tempo d'attesa per confermare la caduta di tensione sull'ingresso: il passaggio da ON a OFF. Questo può essere utile, oltre che per i disturbi, per eliminare i rimbalzi di contatti normalmente aperti. Se si collega un pulsante N.A. tra il +24V e un ingresso, alla pressione del pulsante potrebbero verificarsi dei rimbalzi: inizialmente l'ingresso passa a ON, ma a causa di un cattivo contatto subito dopo diventa OFF. In questo caso (contatto normalmente aperto) si può impostare una soglia XTHRESHOLDDN che introduce stabilità (e un ritardo) nel passaggio da ON a OFF.

XTHRESHOLDUP e XTHRESHOLDDN accettano valori da 0 a 127.

XCOUNT1 e XCOUNT2 (contatore fronti di salita)

Sui primi due ingressi, X1 e X2, sono implementati contatori hardware che contano i fronti di salita; il rispettivo nome è XCOUNT1 e XCOUNT2. Sono contatori a 16: il conteggio va da 0 a 65535, ed è possibile scrivere o azzerare il contenuto. Si noti che questi contatori non dispongono di alcun filtro software e l'inversione della logica non si applica.

Uscite di potenza Y1..Y8

Le uscite di potenza Y1..Y8 si attivano ponendole a ON (o TRUE), per portare l'uscita a 24 volt, e si pongono a OFF (o FALSE) per disattivarle e portare quindi l'uscita a 0 volt. Secondo l'impostazione del flag AUTOUPDATEXY, l'effetto può essere immediato oppure deferito (vedere il paragrafo I/O sincrono o asincrono); in ogni caso le uscite possono essere "lette" - usate come termini di espressioni, e rispecchiano

sempre l'ultima impostazione eseguita, anche se l'aggiornamento della tensione in uscita non è ancora attivo.

YINVERT

Come per gli ingressi, è possibile invertire logicamente un'uscita ponendo la rispettiva YINVERT a ON. Una dichiarazione "DEFINE allarme !Y1" non fa altro che impostare a TRUE la risorsa YINVERT1.

Feedback (allarme uscite) FB12, FB34, FB56, FB78, FBBYTE

EZ-Red è in grado di rilevare, sulle uscite di potenza, il sovraccarico (che porta a un aumento di temperatura) e il circuito aperto (indice eventualmente dell'interruzione di un circuito).

Quando un'uscita Y è bassa (mancanza di tensione), un eventuale circuito aperto viene rilevato, e il rispettivo FB diventa TRUE. Quando un'uscita è attiva (presenza di tensione), un passaggio eccessivo di corrente viene rilevato: l'uscita viene limitata in corrente e, dopo alcuni istanti, il relativo bit FB diventa ON.

La circuiteria di feedback (risposta) gestisce le uscite a coppie: per la coppia Y1+Y2 c'è un singolo bit di feedback, poi un altro bit per Y3+Y4, e altri due per Y5+Y6 e Y7+Y8. La risorsa FBBYTE è il compendio dei quattro feedback (nei primi 4 bit): con le uscite correttamente collegate e configurate è possibile, con un'unica lettura di questo byte, rilevare eventuali guasti elettrici.

Questi feedback possono far intervenire il watch-dog se abilitati tramite WDTFBENAxx (12, 34, 56, 78) o il byte WDTFBBYTE conrrispondenti. Consultare il capitolo del watch-dog per ulteriori informazioni.

TIMER (temporizzatori): TIMERMS, TIMERSEC, TIMERMIN

EZ-Red dispone di 16 timer software da 16 bit, composti da un contatore (conto alla rovescia) e da un contatto. Il contatore può essere letto e scritto normalmente, può contenere valori da 0 a 65535, e si decrementa verso zero autonomamente. Il contatto relativo T, è ON se il timer sta contando (cioé, il suo contatore è superiore a zero), e OFF se il timer ha raggiunto lo zero. Se durante il conteggio un nuovo valore viene scritto nel contatore, il conteggio prosegue dal nuovo valore (cioé, i timer sono *retriggerabili*).

Vi sono 6 contatori al millisecondo: da TIMERMS1 a TIMERMS6, e i relativi contatti d'uscita sono TMS1..TMS6. Scrivendo 1000 dentro TIMERMS1, la sua uscita TMS1 va immediatamente a ON; per il tempo di un secondo (1000 millisecondi), il valore di TIMERMS1 decresce fino a zero, infine il contatto d'uscita TMS1 diventa OFF. Il tempo massimo di questi timer al millisecondo è di circa 65 secondi.

Vi sono poi 6 contatori che contano secondi, da TIMERSEC1 a TIMERSEC6, e i relativi contatti d'uscita sono TSEC1..TSEC6; il loro tempo massimo di conteggio è superiore alle 18 ore. Infine sono disponibili 4 contatori che contano minuti: da TIMERMIN1 a TIMERMIN4, con le relative uscite TMIN1..TMIN4; il tempo massimo di conteggio va oltre i 45 giorni.

I timer sono semplici da capire e usare, specialmente in combinazione con le istruzioni IF. Per esempio, il seguente frammento:

```
define       pulsante     x1
define       spia         y1

Task1:
  if /pulsante timerms1 = 1000
  spia = tms1
```

Fa sì che a ogni pressione del pulsante la spia si accenda per un secondo. Se si preme il pulsante e si mantiene premuto, un singolo lampo viene eseguito. Se non si usa il modificatore del fronte di salita "/", quindi:

```
        if pulsante timerms1 = 1000
```

Il lampeggio si attiva alla pressione del pulsante, e termina un secondo DOPO il rilascio.

Ingressi FX, contatori FXCOUNT e interfaccia encoder

Gli ingressi veloci FX1 ed FX2, oltre che funzionare da normali ingressi (optoisolati, senza negazione software e senza filtri), dispongono di due contatori hardware di 32 bit che contano i fronti di salita.

FXCOUNTLx ed FXCOUNTHx

A ogni fronte di salita del relativo ingresso veloce FX, il contatore s'incrementa di uno; quando raggiunge il valore esadecimale FFFFFFFF (oltre 4 miliardi), il contatore si azzera (overflow). Il contatore può essere impostato a un valore qualsiasi eseguendo un'assegnazione.

I contatori sono a 32 bit, ma la massima dimensione dei dati gestibile da programma è di 16 bit: ogni contatore è quindi spezzato nella sua parte alta (FXCOUNTH) e parte bassa (FXCOUNTL). Per conteggi fino a 65535 la parte alta si può semplicemente ignorare.

Encoder

Gl'ingressi veloci FX1 ed FX2 permettono di collegare i due canali A e B di un encoder in quadratura. In questo caso, senza alcuna ulteriore operazione, EZ-Red tiene traccia della posizione dell'encoder in un registro da 32 bit leggibile da programma. Quando l'encoder esegue un passo in direzione "+", il registro s'incrementa di uno; se il registro già contiene il valore massimo esadecimale FFFFFFFF, si azzera (overflow). Parimenti, quando l'encoder esegue un passo in direzione "-", il registro si decrementa; se è già al valore zero, un decremento lo porta al numero esadecimale FFFFFFFF (underflow).

Il contatore è da 32 bit, ma la massima dimensione dei dati gestibili da programma è di 16 bit, per cui il registro dell'encoder è diviso in due risorse da 16 bit, ENCODERL (la "L" sta per "low") ed ENCODERH (dove la "H" sta per "high"). Se le quote che s'intendo gestire sono limitate a numer da 0 a 65535 è possibile ignorare completamente la parte alta (ENCODERH) e lavorare esclusivamente con ENCODERL.

Preset dell'encoder

E' possibile eseguire un azzeramento o impostazione del registro encoder (preset). Data la discrepanza tra 16 bit (software) e 32 bit (hardware), se si usano entrambe le risorse ENCODERL ed ENCODERH occorre impostare prima la parte bassa e poi la parte alta:

```
; preset encoder
encoderL = 200
encoderH = 0        ; opzionale
```

Appena la parte bassa (encoderL) viene scritta, la parte alta viene posta automaticamente a 1 (in hardware). Ciò non comporta problemi se ENCODERH non viene utilizzata nel ciclo; se invece è utilizzata, l'istruzione successiva la può porre al valore desiderato. Il motivo di questo comportamento è che l'hardware dell'encoder conta continuamente, e potrebbe contare (se l'asse dell'encoder si muove) in un momento intermedio tra la prima assegnazione e la seconda, portando a risultati errati.

Dato che i numeri negativi non sono gestiti, invece di azzerare l'encoder in corrispondenza di un fine corsa o di un micro di zero è preferibile fare il preset con un numero, per esempio 200. Questo consente all'asse di retrocedere ancora un poco senza causare un underflow del contatore hardware.

Controllo di flusso (IF-THEN-ELSE e GOTO)

Per controllo di flusso s'intende la possibilità di modificare lo svolgimento monotono dall'alto verso il basso di un programma. Nel caso di EZ-Red, se non ci fosse il controllo di flusso, ogni task eseguirebbe le proprie istruzioni dall'inizio al fondo, per poi ricominciare: ogni task assomiglierebbe a un ladder, anche se potenzialmente più articolato. Le istruzioni IF permettono di eseguire certe istruzioni (parte THEN) solo se si verificano certe condizioni (parte IF): questo è vicino al modo di pensare degli esseri umani.

Nella forma più semplice, una singola istruzione viene eseguita se una certa condizione è soddisfatta:

```
if x1=on then r2 = on
```

Dopo la parola IF occorre porre un'espressione di tipo boolean (bit), per esempio "ain1>128" o "timerms1<>0". Dopo la parola THEN occorre porre uno statement (istruzione o comando) completo.

E' possibile usare più istruzioni con una singola IF, nel seguente modo:

```
if x1 then
  r2 = on
  dt4 = 128
end
```

Il compilatore riconosce che più istruzioni sono sottoposte alla stessa IF perché la parola THEN non ha termini dopo di essa. Tutte le righe successive, fino all'END, sono eseguite se l'espressione (x1, in questo caso) è TRUE; altrimenti, non sono eseguite.

A volte l'esecuzione condizionata di una parte di codice non è sufficiente: si vorrebbe eseguire un certo codice OPPURE un altro, secondo una condizione:

```
if ain1>128 then aout1=64 else aout1=192
```

Nell'esempio qui sopra, viene eseguito o "aout1=64" oppure "aout1=192", secondo il valore di ain1. La parola ELSE introduce l'istruzione da eseguire se la condizione IF è false. Benché sia possibile e formalmente corretto usare un'istruzione nulla per il THEN:

```
if x1 then else dt1=0
```

simili costrutti sono usati raramente, perché basta invertire la condizione:

```
if not x1 then dt1=0
```

Come per la parte THEN, anche la parte ELSE può essere formata da un blocco di istruzioni:

```
if r1 or x2 then y1=on else
  y1=off
  y2=on
end
```

Infine, come è facile immaginare, possono essere formate da blocchi di istruzioni entrambe le clausole THEN ed ELSE, e ogni blocco può contenere ulteriori istruzioni IF. Il frammento seguente:

```
if temperatura<setpoint then
  res_1000watt = on
  res_2000watt = off
  if temperatura+10 < setpoint then res_2000watt = on
end else
  res_2000watt = off
  if temperatura > setpoint+10 then res_1000watt = off else res_1000watt = on
end
```

è un semplice controllo di forno a due velocità che cerca di mantenere la temperatura in un intervallo stabilito.

Un blocco di istruzioni può contenere quante istruzioni si vogliono, di qualsiasi tipo incluso altre IF, ma non DEFINE o di etichette (che non sono istruzioni, ma *dichiarazioni*). Il numero di IF annidate una dentro l'altra

non ha limite, però il programma potrebbe diventare difficile da leggere.

GOTO (salto incondizionato)

Le istruzioni IF-THEN-ELSE permettono di alterare il normale flusso alto > basso di esecuzione del programma "racchiudendo" una parte di codice. Un altro meccanismo di alterare il flusso, più basilare ma a volte necessario o per lo meno comodo, è quello di usare il salto incondizionato GOTO. Il GOTO trasferisce l'esecuzione a una diversa parte del programma, la quale deve essere etichettata appositamente:

```
IF allarme THEN goto uscita
IF pulsante=off THEN goto uscita

  ; parte di programma complessa
  ...
  ...

Uscita:
  ...
```

L'etichetta di riferimento dell'esempio sopra è "Uscita:", scritta nello stesso modo in cui si immettono le etichette dei Task. Lo stesso frammento di codice potrebbe essere scritto usando blocchi di IF, ma risulta forse meno leggibile:

```
IF allarme=false THEN
  IF pulsante=on THEN
    ; parte di programma complessa
    ...
    ...
  END
END

Uscita:
  ...
```

I salti fatti con GOTO dovrebbero saltare sempre all'interno dello stesso task, non in un altro task o nella sezione iniziale (Init) del programma.

Le etichette devono essere uniche nel programma, esattamente come qualsiasi altro identificatore, però a una singola etichetta possono fare riferimento più GOTO.

Istruzioni e Risorse speciali

Controllo dei task

Un programma EZ-Red può contenere fino a 16 Task. Un task può essere utilizzato, oltre che per controllare parte di un impianto, anche per simulare periferiche come contatori o timer e perfino per simulare delle subroutine. Occorre però avere la possibilità di controllare l'esecuzione di ogni task per fermarlo, riattivarlo, o farlo ripartire dall'inizio.

WAKEUP (attivazione di un task)

Quando un programma (ciclo) viene avviato, tutti i task tranne il numero 1 sono sospesi. In questo modo è possibile preparare una serie di impostazioni (iniziazione) senza che i task paralleli vadano in esecuzione prima che la preparazione sia terminata. Per attivare/risvegliare un task si usa l'istruzione WAKEUP seguita dal numero di task. Il numero di task (da 1 a 16) deve essere espresso con una costante, o un identificatore dichiarato con una DEFINE riferita a una costante:

```
WAKEUP 2     ; attiva il task 2
; oppure...

DEFINE   pid  4
...
wakeup pid
```

Se il task oggetto del WAKEUP è già attivo, l'istruzione non ha effetto; altrimenti, il task riprende l'esecuzione dal punto in cui era stata interrotta. Nella fase iniziale di un programma, tutti i task sono "fermi" in corrispondenza della loro prima istruzione.

RESTART (riavvio di un task)

L'istruzione RESTART è simile a WAKEUP, ma inoltre fa ripartire il task dal suo punto d'ingresso:

```
RESTART 2    ; riarma il task 2 (sospeso o no)
             ; in ogni caso, Task2 riparte dall'inizio

Task3:
  ...
  RESTART
  ...
```

Il punto d'esecuzione del task indicato viene spostato all'inizio e, se il task indicato è fermo (sospeso), viene riattivato. Questa istruzione ha sempre effetto sul task indicato, perché in ogni caso lo fa ripartire dall'inizio.

L'argomento di RESTART è opzionale: se omesso, è implicito il task corrente. Chiaramente, un task sospeso non è in grado di auto-applicarsi un RESTART.

SUSPEND

Un task può essere anche sospeso con l'istruzione SUSPEND. Il task indicato viene bloccato (sospeso) nel punto in cui si trova, e un successivo WAKEUP lo farà riprendere dallo stesso punto. L'argomento di SUSPEND può essere omesso per indicare il task corrente, ed è il modo migliore per sospendere un task perché la sospensione avviene in un punto definito. In altre parole, è meglio che un task sospenda sè stesso piuttosto che venga sospeso da qualche altro task, rimanendo in una posizione sconosciuta:

```
; il task 2 genera un impulso ritardato
Task2:
  wait 1000         ; attesa di un secondo
  Y1 = ON
  wait 500
  Y1 = OFF
  suspend

Task3:
  ; per generare un impulso ritardato, basta eseguire:
  wakeup 2
```

Il frammento qui sopra mostra come, con semplicità, sia possibile generare un impulso ritardato e concorrente al task corrente senza neppure usare un timer.

CYCLERUN

Il bit CYCLERUN è ON mentre il programma PLC è in esecuzione: esiste principalmente per dare modo al PC collegato di sapere se il ciclo è in funzione. Però, il ciclo PLC può leggere questo bit (e naturalmente il contenuto risulta TRUE/ON), e può anche scriverlo. Se da ciclo viene impostato a OFF il ciclo stesso viene fermato.

Watch-dog

Il Watch-dog (letteralmente, “cane da guardia”) è un meccanismo attraverso il quale si tengono sotto controllo certi aspetti di un sistema per determinare se vi siano anomalie; nel caso eventuale che una o più anomalie fossero rilevate, il sistema viene posto in uno stato sicuro (uno stato in cui non si creano danni ulteriori).

Nel caso di un controllo di automazione, per esempio il controllo di temperatura di un forno, lo stato insicuro è costituito dal forno acceso: se il sistema di regolazione s'interrompesse il forno potrebbe raggiungere temperature troppo elevate, oltre al fatto che la lavorazione (cioé, lo scopo per cui si usa un forno) si fermerebbe.

Nel caso di EZ-Red, il watch-dog può essere configurato per controllare i seguenti aspetti del sistema:

1) la comunicazione con il PC (se cessa, significa che una parte del sistema non funziona)
2) il funzionamento delle uscite di potenza (sovraccarico e circuito aperto)
3) le alimentazioni interne di EZ-Red

Se configurato per controllare uno o più degli aspetti descritti, e se una anomalia viene rilevata, il watch-dog interviene e provoca quanto segue:

1) pone le uscite di potenza in una condizione precisa (configurabile), considerata “sicura”
2) imposta le uscite analogiche a un determinato livello (configurabile)
3) ferma l'esecuzione del ciclo PLC (opzione escludibile)

Quanto esposto qui sopra avviene 100 volte al secondo: se il PC riconfigura le uscite mentre il watch-dog è attivo, entro il centesimo di secondo successivo il watch-dog le reimposta.

Se l'opzione di arresto del ciclo (WDTSTOPSCYCLE) è spenta, e un ciclo è in esecuzione, allora l'unico effetto del watch-dog è quella di alzare il bit relativo di intervento (WDTFIRED); le uscite non vengono toccate perché in tale modo di funzionamento è responsabilità del ciclo PLC o del computer intraprendere le opportune misure per correggere il problema.

WDTFIRED e WDTSTOPSCYCLE

WDTFIRED è un bit che segnala se il watch-dog è attivo, ed è read-write: scrivendo TRUE o FALSE è

possibile, rispettivamente, forzare l'intervento del watch-dog o cancellarlo. La cancellazione del watch-dog è quasi inutile se le condizioni per il suo intervento persistono: occorre prima correggere le cause, o disabilitare i controlli che rilevano le dette cause.

WDTSTOPSCYCLE è un bit che indica se l'intervento del watch-dog comporta l'arresto del ciclo PLC. Il valore di default, all'accensione, è TRUE – cioé se il watch-dog interviene, il ciclo in esecuzione viene fermato. Se WDTSTOPSCYCLE è impostato a false, allora il ciclo non viene interrotto e anzi il ciclo stesso può rilevare l'intervento del watch-dog e prendere determinate decisioni conseguenti.

WDTOUTS, WDTAOUT1-2 (configurazione delle uscite del Watch-dog)

Queste tre risorse contengono la configurazione desiderata delle uscite in caso d'intervento del watch-dog. WDTOUTS è un BYTE che, esattamente come YBYTE, contiene un bit per ogni uscita di potenza. Mentre il watch-dog è attivo, in effetti, WDTOUTS viene copiato continuamente su YBYTE. Lo stesso viene fatto per le due uscite analogiche: WDTAOUT1 viene continuamente copiata su AOUT1, e WDTAOUT2 su AOUT2.

Il valore di default per WDTOUTS è 192 (le uscite da 1 a 6 spente, la 7 e la 8 accese), e quello per WDTAOUT1 e WDTAOUT2 è 0.

WDTTIME (timeout della comunicazione con il PC)

Questa è una risorsa di tipo INTEGER e contiene un numero, da 0 a 65535, che indica il timeout in millisecondi della ricezione di comandi dal PC. Se un timeout di comunicazione viene rilevato, il watch-dog interviene. Per esempio, impostando il valore 1000, EZ-Red si aspetta una richiesta da PC, attraverso la porta USB, almeno una volta al secondo (1000 millisecondi). Se anche una sola richiesta arriva dopo un secondo da quella precedente, il watch-dog interviene. Il valore corretto del timeout dovrebbe essere calcolato in base alle esigenze dell'applicazione, ma tenendo presente che i personal computer, in certe condizioni, possono risultare poco responsivi: un timeout di un secondo potrebbe essere troppo piccolo per un sistema operativo grafico a finestre.

Il valore 0 (default all'accensione) esclude il controllo sulla comunicazione.

WDTFBENA12-34-56-78 e WDTFBBYTE (controllo uscite di potenza)

Queste risorse booleane (o il loro compendio nei primi 4 bit di WDTFBBYTE) collegano i feedback delle uscite di potenza al watch-dog: se un feedback (FB12, FB34...) diventa TRUE, e la relativa abilitazione del watch-dog (WDTFBENA12, WDTFBENA34...) è anch'essa TRUE, il watch-dog interviene.

I bit di feedback rilevano un eventuale consumo eccessivo di corrente quando l'uscita è ON, e un eventuale circuito interrotto quando la corrispondente uscita Y è OFF. Le uscite Y che non sono collegate dovrebbero essere mantenute alte (ON), se non si vuole che il bit di feedback diventi TRUE; è anche possibile ignorare tale bit di feedback e non abilitare la relativa abilitazione del watch-dog.

I bit di feedback (FB12, FB34...) e le corrispondenti abilitazioni del watch-dog si riferiscono a coppie di uscite (1-2, 3-4, 5-6, 7-8).

Interazione tra Watch-dog, PC e ciclo PLC

Come accennato prima, se il watch-dog è attivo la configurazione “sicura” delle uscite viene impostata 100 volte al secondo, anche se il PC continua a modificarle. Il PC dovrebbe correggere le cause che scatenano il watch-dog, oppure disabilitare la rilevazione dei guasti (WDTTIME e/o WDTFBBYTE) prima di spegnere il bit WDTFIRED.

Se c'è un ciclo in esecuzione, e WDTSTOPSCYCLE è FALSE, l'intervento del watch-dog non ferma il ciclo e lo

stesso ciclo può (o deve) prendere provvedimenti: è sua responsabilità rilevare lo stato di emergenza ed eseguire le azioni opportune (perché le uscite NON vengono modificate automaticamente). Fra i tanti modi possibili, uno può essere quello di riservare un task apposito, come in questo esempio:

```
Task3:
  ; gestione dell'intervento del watch-dog
  WAIT WDTFIRED     ; attesa intervento watch-dog

  ; se arriva qui, è perché il WDT è intervenuto
  SUSPEND 1         ; fermare i task che modificano le uscite
  SUSPEND 2
  ...  ; altre azioni
  YBYTE = 128       ; impone un configurazione delle uscite “sicura”
  ...  ; altre azioni
```

Il Task3 attende (WAIT) che il watch-dog intervenga – cosa che naturalmente può non avvenire mai. Se il watch-dog interviene, con minima latenza il Task3 riprende la sua esecuzione e, in questo esempio, per prima cosa sospende i task che potrebbero interferire; infine pone le uscite in una configurazione considerata sicura (attuatori spenti, un segnalatore di emergenza sull'uscita 8 acceso).

Naturalmente in fase di iniziazione occorre attivare il task 3 con "WAKEUP 3".

Istruzione LOG (acquisizione dati e debug)

Quando un PC è collegato con EZ-Red, può acquisire e intervenire in vari modi su ingressi, uscite e dati interni. Però i tempi di latenza del PC, se non ha un sistema operativo real-time, sommati con la latenza dell'interfaccia USB, possono diventare eccessivi; è per questo motivo che il dispositivo implementa un ciclo programmabile: per avere tempi di reazione più rapidi e predicibili. Nel caso in cui si voglia fare un'acquisizione dati in corrispondenza di un certo evento, EZ-Red la può eseguire con una ragionevole velocità, ma non sempre altrettanto può fare il PC a causa della latenza prima citata. In questi casi l'istruzione LOG è utile: EZ-Red esegue l'acquisizione di un valore, e la comunica al PC usando appunto LOG che si avvale di un buffer circolare (memoria di transito) per poter accumulare alcuni eventi e sopperire ai tempi di latenza. L'istruzione LOG serve a inviare dati al PC, un integer di 16 bit o un singolo bit, su un “canale” a scelta fra 255. La sintassi dell'istruzione è:

```
LOG canale valore_integer_o_bit
```

Il PC, collegato via USB e usando le funzioni della DLL (message di windows) oppure il modo terminale seriale, riceve una comunicazione asincrona che non interferisce con il dialogo in corso. Questa funzione può essere utile anche per aiutare la messa a punto di un programma.

REPORTBACK, SENDTOPC (trasmissione degli ingressi)

Il bit REPORTBACK, se impostato a TRUE, invia automaticamente una comunicazione al PC a ogni variazione degli ingressi digitali; nel pacchetto inviato vi è anche la posizione dell'encoder. La stessa comunicazione può essere eseguita programmaticamente impostando a ON il bit SENDTOPC, che viene reimpostato a OFF automaticamente a trasmissione avvenuta.

Funzioni speciali di configurazione, protezione, diagnostica

EZ-Red è dotato di alcuni ulteriori meccanismi che possono essere usati da ciclo a fini di protezione e di diagnostica.

CONFIGxxx (BIT, CHR, WRD)

EZ-Red dispone di 16 integer a 16 bit (CONFIGWRD1..16) non volatili: il loro contenuto viene preservato tra un'accensione e l'altra perché viene scritto nella memoria flash interna del dispositivo.

Per comodità di gestione, questi 16 integer sono accessibili anche in byte: le risorse CONFIGCHR1..32 sono sovrapposte fisicamente a CONFIGWRDx, e la modifica di una si riflette immediatamente sull'altra. La disposizione è little-endian, cioé CONFIGWRD1 contiene nella parte bassa CONFIGCHR1, e nella parte alta CONFIGCHR2; WRD2 contiene CHR3 e CHR4, WRD3 le CHR5 e 6, e via dicendo. Scrivendo CONFIGWRD1=1, implicitamente CONFIGCHR1 diventa 1 e CONFIGCHR2 diventa 0.

Le prime 4 risorse CONFIGCHR1..4, corrispondenti pure a CONFIGWRD1..2, sono anche indirizzabili come singoli bit, con i nomi CONFIGBIT1..32. La seguente tabella mostra nel dettaglio la mappa della memoria:

INT (16 bit)	BYTE (8 bit)	BIT
CONFIGWRD1	CONFIGCHR1	CONFIGBIT1 ... fino a ... CONFIGBIT8
	CONFIGCHR2	CONFIGBIT9 ... fino a ... CONFIGBIT16
CONFIGWRD2	CONFIGCHR3	CONFIGBIT17 ... fino a ... CONFIGBIT24
	CONFIGCHR4	CONFIGBIT25 ... fino a ... CONFIGBIT32
CONFIGWRD3	CONFIGCHR5-6	NON INDIRIZZABILI COME BIT
CONFIGWRD4	CONFIGCHR7-8	
CONFIGWRD5	CONFIGCHR5-6	
CONFIGWRD6	CONFIGCHR5-6	
CONFIGWRD7	CONFIGCHR5-6	
CONFIGWRD8	CONFIGCHR5-6	
CONFIGWRD9	CONFIGCHR5-6	
CONFIGWRD10	CONFIGCHR5-6	
CONFIGWRD11	CONFIGCHR5-6	
CONFIGWRD12	CONFIGCHR5-6	
CONFIGWRD13	CONFIGCHR5-6	
CONFIGWRD14	CONFIGCHR5-6	
CONFIGWRD15	CONFIGCHR5-6	
CONFIGWRD16	CONFIGCHR5-6	

BCW_xxx (BIT, CHR, WRD)

Queste risorse speciali servono a manipolare bit, byte (char) e word (di 16 bit) senza usare troppe operazioni logiche. Come CONFIGXXX, più risorse condividono lo stesso spazio di memoria, permettendo di scrivere un dato in un modo e leggerlo in un altro. Vi sono 2 interi di 16 bit, BCW_WRD1 e BCW_WRD2, che sono visibili anche come BCW_CHRx (da 1 a 4) e BCW_BITx da 1 a 32.

DISABLEUSB

Se questo bit è impostato (ON o TRUE), EZ-Red rifiuta la comunicazione USB a eccezione delle richieste di:

- identificazione (modello e versione firmware)

- lettura e scrittura dei parametri

- sblocco (con password, se impostata)

FIRSTRUN

Questo bit indica se il Task1 sta eseguendo la scansione per la prima volta (quando TRUE) o se ha eseguito almeno una volta un salto all'indietro (FALSE). La sua esistenza è mutuata dal paradigma dei PLC, ma

nell'architettura di EZ-Red non è necessario perché è più comodo e corretto usare la parte INIT (iniziazione del programma PLC).

FORCEDXS

Il PC, attraverso una funzione speciale, può forzare virtualmente uno degli ingressi digitali in un certo stato; in queste condizioni EZ-Red si comporta come se realmente leggesse dall'ingresso lo stato virtuale invece di quello reale. Il bit FORCEDXS indica che uno o più ingressi sono nello stato forzato, e il programma PLC può comportarsi diversamente a seconda dello stato di questo bit.

PCCONNECTED

Questo bit indica al programma PLC che un PC è collegato sulla porta USB e la comunicazione è instaurata. Questo bit diventa FALSE dopo 60 secondi dall'ultima ricezione valida sulla porta USB.

PWDPROTECT

Questo bit segnala, se ON, che una password di protezione è stata impostata per proteggere le modifiche al programma PLC. Quando PWDPROTECT è attivo, il programma non può essere variato o memorizzato nella flash interna. Da programma è possibile disattivare il bit (ma non la password): la protezione, se disattivata in questo modo, verrà abilitata nuovamente al prossimo riavvio (ciclo di alimentazione). Di converso, è possibile per il programma attivare questo flag anche se nessuna password è realmente impostata: EZ-Red si comporterà come se fosse protetto, fino al prossimo riavvio.

Esempi di programmi

Alcuni programmi d'esempio possono essere d'aiuto per vedere in concreto le caratteristiche del linguaggio.

Gli esempi che seguono sono riportati a scopo didattico, e non hanno la pretesa di essere corretti o scritti nel migliore modo possibile.

Pressa con controllo di sicurezza (procedurale)

In questa applicazione una pressa viene comandata da due pulsanti; il ciclo può iniziare solo se entrambi risultano premuti dopo che entrambi sono stati rilasciati. I pulsanti sono del tipo normalmente aperto e, quando premuti, portano tensione al rispettivo ingresso digitale che diventa ON. Per evitare rimbalzi si usano le risorse XTHRESHOLDxx; dato che i pulsanti sono normalmente aperti, è importante evitare che un cattivo contatto possa essere interpretato come un rilascio, quindi il tempo "DN" (XTHRESHOLDDN) è quello più importante.

Questo esempio ha forma procedurale. Vi sono diversi modi per scrivere lo stesso programma in modo procedurale, con meno IF e GOTO, e anche in modo funzionale.

```
; Controllo pressa

define pressa  Y3
define puls1   x1
define puls2   x2

XTHRESHOLDUP1 = 40
XTHRESHOLDUP2 = 40
XTHRESHOLDDN1 = 100
XTHRESHOLDDN2 = 100

Rilascia:
  pressa = OFF
  if puls1 or puls2 then goto rilascia

Attendi:
  if puls1 and puls2 then goto attiva
  goto attendi

Attiva:
  if not puls1 then goto rilascia
  if not puls2 then goto rilascia
  pressa = ON
  goto attiva
```

Slitta con pulsante e due fine corsa (funzionale)

In questa applicazione c'è una slitta mossa da un motore, nelle direzioni avanti e indietro. Due fine corsa delimitano il movimento. La slitta è, a riposo, ritirata completamente – sul fine corsa indietro. Un pulsante consente di muovere la slitta in avanti, fino al fine corsa; appena il comando cessa, la slitta deve tornare indietro. Se il comando (la pressione del pulsante) dura più di 10 secondi, la slitta deve ritirarsi comunque.

Il programma viene fornito in due versioni, una *funzionale* e una *procedurale*.

La versione funzionale è una sequenza di assegnazioni, in modo simile a un normale ladder. La prima parte del programma definisce in modo mnemonico tempi e risorse utilizzati:

```
; Comando slitta

define max_tempo 10000    ; tempo di validità del comando

define ok_indietro  x1    ; NON-fine-corsa indietro
define ok_avanti    x2    ; NON-fine-corsa avanti
define pul_avanti   x3    ; pulsante di movimento avanti
define mot_avanti   y1    ; slitta in avanti
define mot_indietro y2    ; slitta indietro

define timer timerms1
define com_valido   tms1  ; timer di validità comando
  ; indica che il comando del pulsante non ha superato
  ; i 10 secondi
```

Questa prima parte del programma assegna semplicemente nomi chiari alle risorse. La parte rimanente di programma esegue il ciclo:

```
; il comando può durare al massimo 10 secondi (10000 ms)
if /pul_avanti then timer = max_tempo
if !pul_avanti then timer = 0
  ; ora com_valido indica se il comando è valido

; andare avanti se comando valido
; e se il fine corsa non è ingaggiato (ok_avanti)
mot_avanti = com_valido and ok_avanti

; andare indietro se non c'è comando o è scaduto
; fino al fine corsa indietro
mot_indietro = ok_indietro and not com_valido
  ; oppure: mot_indietro = ok_indietro and !com_valido
```

Il programma è funzionale perché non esistono vere istruzioni IF; l'unica usata è quella che serve per caricare il timer, e usa la notazione del fronte di salita: è la diretta trasposizione del ramo di un ladder che attiva un timer sul fronte di salita di un contatto.

Il programma descritto risponde effettivamente ai requisiti enunciati prima: "il comando deve durare al massimo 10 secondi". Però, scritto così, è possibile rilasciare brevissimamente il pulsante e ripremerlo per avere un comportamento che potrebbe essere indesiderabile. Quando la slitta arretra e il pulsante è rilasciato, una pressione del pulsante fa ripartire la slitta in avanti. Potrebbe essere desiderabile che la slitta torni indietro completamente prima di poterla riavviare in avanti. Per fare questo è sufficiente cambiare l'istruzione

```
if !pul_avanti then timer = 0
```

con:

```
if mot_indietro or !pul_avanti then timer = 0
```

Slitta con pulsante e due fine corsa (procedurale)

Il programma che segue è la versione procedurale, che analizza le varie fasi del ciclo piuttosto che esprimere lo stato di ogni uscita a prescindere dal momento (contesto) del ciclo. Date quase le stesse definizioni del programma precedente (il timer non serve più):

```
; Comando slitta
define max_tempo 10000    ; tempo di validità del comando

define ok_indietro  x1    ; NON-fine-corsa indietro
define ok_avanti    x2    ; NON-fine-corsa avanti
define pul_avanti   x3    ; pulsante di movimento avanti
define mot_avanti   y1    ; slitta in avanti
define mot_indietro y2    ; slitta indietro
```

il programma descrive una per volta le varie fasi:

```
; A riposo, la slitta deve arretrare fino al fine-corsa
Riposo:
  mot_avanti = OFF
  mot_indietro = ok_indietro
  if pul_avanti and !mot_indietro then goto avanti
  goto riposo

Avanti:
  mot_indietro = OFF
  mot_avanti = ON

  ; attendere: scadenza, o fine corsa, o fine comando
  wait max_tempo !ok_avanti !pul_avanti
  if waitremain1=0 then goto riposo    ; tempo scaduto
  if !pul_avanti then goto riposo      ; pulsante rilasciato
    ; pulsante premuto e tempo ancora disponibile

  ; attendere fino al fine corsa per il tempo rimanente
  wait waitremain1 !ok_avanti !pul_avanti
  if waitremain1=0 then goto riposo    ; tempo scaduto
  if !pul_avanti then goto riposo      ; pulsante rilasciato

  ; arrivato sul fine corsa
  mot_avanti = OFF
  wait waitremain1 !pul_avanti

  goto riposo
    [ questo GOTO non servirebbe, il ciclo riparte
      automaticamente ]
```

Il programma è più lungo, ma analizza in maggior dettaglio tutte le singole fasi.

Slitta, procedurale, utilizzando due task

Utilizzando in modo creativo il multitasking è possibile scrivere il programma in modo diverso. Assumendo che il rilascio del pulsante di comando deve ritirare la slitta, un task può occuparsi di attendere questo evento e riarmare (RESTART) il Task1. Il listato diventa lineare e molto comprensibile:

```
[ usare le dichiarazioni precedenti ]

Task1:
  mot_avanti = OFF
  mot_indietro = ON
  wait !ok_indietro
  mot_indietro = OFF
  wait pul_avanti

  mot_avanti = ON
  restart 2  ; sorveglia il rilascio del pulsante

  ; attendere scadenza o fine corsa
  wait max_tempo !ok_avanti
  mot_avanti = OFF
  wait waitremain1
  restart

Task2:
  wait !pul_avanti
  restart 1
  suspend    ; evitare di riarmare Task1 in continuazione!
```

Controllo motore con encoder

In questa applicazione si vuole pilotare un motore con i comandi avanti e indietro, e con velocità regolata tramite un comando 0-10 volt. Quando si preme il pulsante di avvio ciclo, il motore esegue una tratta in avanti fino a una quota predeterminata o fino al rilascio del pulsante. Quando il pulsante è rilasciato, il motore torna indietro fino al fine corsa.

La prima parte del programma dichiara nomi mnemonici per il cablaggio di ingressi e uscite:

```
; Controllo motore con encoder

define quota        28414        ; arbitraria (esempio)

define mot_avanti   y1     ; comando "motore avanti"
define mot_indietro y2     ; comando "motore indietro"
define start        x1     ; pulsante d'avvio ciclo
define finecorsa    !x2    ; fine corsa (off=finecorsa)
define speed        aout1  ; uscita 0-10V per velocità
```

La seconda parte è l'iniziazione: si esce dal fine corsa in bassa velocità, e poi si rientra:

```
Init:
  ; andare fuori dal micro di fine corsa
  speed=10
  mot_avanti = ON;
  wait 5000 !finecorsa          ; uscire dal fine corsa, o timeout
  if finecorsa then goto errore  ; non è riuscito a uscire da F.C.

  ; tornare indietro per azzerare sul micro
  mot_avanti = OFF
  mot_indietro = TRUE
  wait 15000 finecorsa
  encoderl=200                  ; preset della quota a 200
  mot_indietro = OFF
  if finecorsa then goto Ciclo   ; ok, si può procedere

Errore:
  wdtfired = ON;    ; questo ferma tutto
```

La parte principale del programma si occupa solo di mandare il motore avanti e indietro, secondo lo stato del pulsante d'avvio. La regolazione della velocità viene delegata al task numero 2; se il pulsante d'avvio viene rilasciato prima dell'arresto del motore in quota, è necessario invertire la marcia dolcemente; il rallentamento viene fatto dal task principale, per mostrare un modo alternativo di eseguire una rampa:

```
Task1:
Ciclo:
  mot_indietro = OFF
  wait start                    ; attendere pulsante start

  speed=10                      ; andare avanti, con rampa
  mot_avanti = ON
  restart 2                     ; il task 2 cura rampa e arresto
  wait not start
  suspend 2

  ; tornare indietro, con inversione dolce
 rallenta:
  if speed > 10 then            ; diminuisce la velocità fino a 10
    speed = speed-1
    wait 10
    goto rallenta
  end

  mot_avanti=off
  mot_indietro=on

  wakeup 2
  wait finecorsa
```

Il task numero 2 esegue le rampe d'avviamento e d'arresto. Si noti che mentre il task 2 è in esecuzione, il task 1 non viene arrestato, ma attende il rilascio del pulsante. Il task numero 2 è il seguente:

```
Task2:
  ; va fino alla quota desiderata (quota) o 200 (zero)
  ; se vicino all'arrivo, diminuisce la velocità; se no, la aumenta
  if mot_avanti then dt1 = quota – encoderL else
    dt1 = encoderL
    if dt1 > 200 then dt1 = dt1-200
  end
  if dt1 < 900 then
    speed = dt1 / 5 + 5          ; vicini all'arrivo - rallentare
    if dt1<5 then
      if mot_avanti then mot_avanti=off       ; se motore avanti, fermare
    end
  end else
    if speed<200 then            ; se possibile, aumentare
       speed=speed+5
       wait 5
    end
  end
```

XON ELECTRONICS SRL
WWW.XONELECTRONICS.IT
INFO@XONELECTRONICS.IT

Pagina internet del prodotto: http://www.xonelectronics.it/prodotti/industriali/EZ-Red

Si prega di segnalare errori o imprecisioni a web@xonelectronics.it