EZ-Red - Modulo I/O di potenza

Manuale di programmazione

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
- 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

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 autonoma. E' l'ideale per cicli semplici o complessi dove non è richiesta una interfaccia utente, oppure questa interfaccia è semplice, cioé costituita da spie, interruttori e potenziometri gestibili direttamente dal ciclo macchina.

2. Esecuzione combinata EZ-Red + Computer; il computer esegue solo 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 del ciclo più complesso.

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. Se si usa un editor esterno, esso deve generare file di testo semplici: i word processor non sono adatti. Il testo del programma deve essere compilato, trasferito al modulo EZ-Red, e messo in esecuzione; queste tre fasi sono tutte eseguite usando TSmon. Quando si è soddisfatti del risultato, il programma va memorizzato (store) nella memoria non volatile (flash) del modulo, altrimenti all'accensione successiva EZ-Red carica il programma precedente (l'ultimo 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 per certi versi al BASIC, soprattutto per il fatto che è *procedurale* e *imperativo*. Imperativo significa che le operazioni da svolgere sono comandate con verbi come GOTO, SUSPEND, WAIT; procedurale significa che i vari verbi vengono eseguiti in una sequenza precisa - quella descritta dal testo del programma. Ingressi e uscite del modulo vengono assimilati a variabili che possono essere lette e scritte: un comando come "Y1=1" 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.

Estensioni specifiche: DEFINE e TASK

Il linguaggio comprende due estensioni volte ad aiutare la stesura del ciclo (programma). La prima riguarda la parola chiave DEFINE, che permette di assegnare un nome a una risorsa hardware (ingressi, uscite, variabili). 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 di un ciclo è quindi possibile scrivere "MOTOR=ON" e "MOTOR=OFF" per accendere e spegnere il motore collegato all'uscita 1. Anche ON e OFF sono due identificatori, creati con DEFINE, dichiarati automaticamente da TSMON.

La seconda estensione riguarda il concetto di *Task* (compito). Nei normali PLC programmati con il linguaggio Ladder, ogni ramo del ladder viene valutato (eseguito) di continuo, ed è sostanzialmente un Task - un compito da eseguire contemporaneamente e indipendentemente dagli altri. Il vantaggio del ladder è che rappresenta in sostanza uno schema elettrico; lo svantaggio è che è difficile, con uno schema elettrico, esprimere passaggi successivi di un ciclo. Uno schema elettrico (e quindi il ladder) è un linguaggio funzionale, dove le operazioni descritte avvengono tutte contemporaneamente; il linguaggio EZ-Red è invece procedurale: le operazioni vengono eseguite in sequenza una dopo l'altra. Lo svantaggio è che esso descrive una sola operazione per volta e, durante l'esecuzione dell'operazione o procedura (sequenza di operazioni), le altre operazioni sono sospese. Se in un ladder si scrive l'equivalente di "Y1=X1", si è sicuri che, in qualsiasi momento, quando cambia l'ingresso 1 cambia anche l'uscita 1. In un linguaggio procedurale, "Y1=X1" viene eseguito solo quando incontrato nel testo, e questo potrebbe accadere una volta sola, se il testo indica così.

Con EZ-Red è possibile definire fino a 16 Task, eseguiti tutti contemporaneamente. In questo modo è possibile assegnare a un task il compito "Y1=X1" senza tuttavia abbracciare la filosofia funzionale del linguaggio ladder. I task, inoltre, possono essere sospesi e riavviati. Per capire meglio come l'uso dei task possa essere utile, si supponga di dover realizzare due onde quadre con diversi periodi sulle uscite 1 e 2; siano i periodi 5 e 8 secondi. Questo è un compito facile in un ladder, ma è difficile in linguaggio procedurale. Con il linguaggio EZ-Red si può scrivere quanto segue:

```
Task1:
  y1=on
  wait 3000
  y1=off
  wait 2000

Task2:
  y2=on
  wait 4000
  y2=off
  wait 4000
```

La descrizione 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 5000 millisecondi, poi la spegne; attende ancora 5000 millisecondi e poi, implicitamente, ricomincia. Il risultato è un'onda quadra con periodo di 5 secondi. 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.

I/O: asincrono (automatico) o sincrono (programmatico)

I classici PLC aggiornano gli I/O una volta per ciclo (in modo *atomico*): 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*" è difficilmente applicabile all'EZ-Red, per via del suo approccio procedurale e della presenza di task concorrenti. Quindi, normalmente, ingressi e uscite sono aggiornati automaticamente al ritmo di 1000 volte al secondo: un tempo abbastanza rapido per qualsiasi applicazione di controllo industriale. Tuttavia questo aggiornamento asincrono può essere fonte di problemi, se è richiesto un aggiornamento *atomico*. Si consideri il seguente esempio, dove i comandi "mot_avanti" e "mot_indietro" sono collegati a due relé che alimentano un motore invertendo la polarità per invertire il senso si marcia:

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

I due relé non devono mai essere attivi contemporaneamente, perché se lo sono provocano un corto circuito sull'alimentazione; sovente in questi casi si usano contatti incrociati fra i due relé come misura precauzionale. Come si vede dal programma, i due comandi sono sempre l'uno opposto all'altro.
Il programma ladder equivalente, scritto in un PLC, non comporta alcun problema perché gli I/O vengono aggiornati (resi effettivi) solo alla fine del ciclo. In EZ-Red, invece, gli I/O sono normalmente asincroni: un aggiornamento potrebbe verificarsi esattamente dopo la prima istruzione ma prima della seconda, e per un tempo di 1 millisecondo i due relé potrebbero essere attivi contemporaneamente (per la verità, difficilmente il problema si porrebbe con veri relé - il tempo di un ms è troppo breve per farli muovere). Il problema può essere risolto usando l'approccio procedurale invece che funzionale; l'esempio che segue:

```
; muovere il motore avanti o indietro
if avanti then
  mot_indietro = OFF
  mot_avanti   = ON
end else
  mot_avanti   = OFF
  mot_indietro = ON
end
```

fa esattamente la stessa cosa, senza mai generare la sovrapposizione delle due uscite.
EZ-Red dispone di un ulteriore meccanismo per ovviare al problema, nel caso in cui non si voglia rinunciare alla

programmazione funzionale. E' possibile sospendere, per un tempo limitato o per sempre, l'aggiornamento asincrono degli I/O; questo si fa ponendo "SYNC_IO = ON". Quando vi sono spezzoni di codice funzionale che richiedono atomicità, questi spezzoni possono essere circondati da una sospensione temporanea dell'I/O asincrono, nel seguente modo:

```
; muovere il motore avanti o indietro
SYNC_IO = TRUE      ; disabilitare temporaneamente
mot_avanti   = avanti
mot_indietro = NOT avanti
SYNC_IO = FALSE     ; riabilitare
```

Il frammento esposto dovrebbe risultare familiare a chi conosce gli *interrupt*. A tutti gli effetti, 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.
In alternativa, se buona parte del programma usa l'approccio funzionale, si può tenere sempre disabilitato l'I/O asincrono, e usare il comando UPDATEIO (almeno una volta per ciclo). Attenzione: se SYNC_IO è ON, l'istruzione WAIT non è in grado di leggere correttamente gli ingressi fisici X1..X8 e FX1/FX2, a meno che un altro task non usi UPDATEIO.

Nel capitolo degli esempi, il primo di questi mostra più in dettaglio queste tecniche.

Tipi di dato e Risorse

Molti linguaggi di programmazione hanno il concetto di tipo di dato; alcuni linguaggi permettono di definire nuovi tipi. EZ-Red possiede solo due tipi: i numeri interi, da 0 a 65535, e i valori booleani (BIT), che hanno solo il valore 1 o 0 (on/off, acceso/spento, 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. Nel controllo di flusso si possono usare solo i tipi BIT, ma attraverso gli operatori di confronto (diverso, minore, maggiore...) è possibile ricavare un valore booleano.

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 questo caso, l'espressione X1 è di tipo BIT, e quindi è compatibile con Y1 - anch'essa di tipo bit. In teoria sarebbe un errore scrivere "Y1=AIN1", poiché AIN1 è di tipo *numero*; in realtà il compilatore converte il numero AIN1 in un BIT, confrontando il valore con zero. E' invece sempre un errore scrivere "AIN1=Y1", perché in questo caso i due tipi sono diversi, e il compilatore non cerca di convertire un BIT in un numero. Riferirsi al paragrafo Espressioni per maggiori informazioni.

Struttura del programma

Come visto, il programma può essere suddiviso in task, ma non è obbligatorio; ogni task inizia con 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. 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. Se non vengono dichiarati task, il testo inserito costituisce implicitamente il Task1, e non esiste alcuna sequenza di iniziazione. Il caso più semplice è il seguente, senza iniziazione e senza task:

```
y1=x1
```

L'istruzione y1=x1 viene eseguita all'avviamento, e ripetuta per sempre. Per avere un'iniziazione e un ciclo ripetitivo:

```
y2=ON

Task1:
 y1=x1
```

L'istruzione y2=ON viene eseguita solo all'avviamento, mentre y1=x1 viene eseguito per sempre.
La parte di iniziazione può essere identificata con un'etichetta, per poterle passare il controllo:

```
Init:
  y2=ON
  wakeup 2

Task1:
  y1=x1
  wait 15000
  goto init

Task2:
  y2=x2
  wait 20000
  goto init
```

L'esecuzione di Task1 ricomincia da INIT dopo 15 secondi. 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. La particolarità di questo esempio è che anche il task 2, dopo 20 secondi, ricomincia da Init. Alla fine ci saranno due task che eseguono lo stesso pezzo di codice, e ciò non è raccomandato: un salto all'inizio del programma dovrebbe essere eseguito solo da Task1.

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:

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

y1=x1        ; anche questo è un commento
```

Identificatori e numeri

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.

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 attualmente 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 a Y2
DEFINE fine_corsa   x1
DEFINE tempo_attesa 2000   ; definisce una COSTANTE
```

Istruzioni (statement)

Le istruzioni differiscono dalle dichiarazioni perché generano effettivamente codice per il PLC di EZ-Red. Vi sono tre tipi di istruzioni: *assegnazioni*, *comandi*, e i comandi di *controllo di flusso*. In molti casi si possono usare *espressioni*, che sono una combinazione di identificatori (*risorse*) 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.

Espressioni

Un'espressione è un calcolo aritmetico, come "1+5", oppure un calcolo logico, come "x1 OR x2".

Tutte le espressioni devono iniziare con un nome di risorsa o una costante, e proseguire con coppie di "operatore risorsa". Le espressioni sono calcolate da sinistra a destra, senza precedenze: il risultato di 1+2*4 (uno più due per quattro) è 12, non 9 come nell'aritmetica classica. Ogni operando impone quali operatori possono trovarsi alla sua destra. Per esempio, se l'operando è di tipo numero, può essere seguito dall'operatore "+"; ma se l'operando è di tipo bit, non può essere seguito da "+". Alcuni operatori (AND, XOR, =...) esistono nella doppia versione per bit e per numeri; in tal caso, l'operando di sinistra impone la versione da usare.

A sua volta, un operatore impone il tipo di operando alla sua destra. Così, l'operatore "+" richiede, alla sua destra, un operando di tipo numero. Gli operatori che esistono in due versioni, per bit e per numeri, come AND e OR, richiedono lo stesso tipo di operando a sinistra e a destra.

Per quanto esposto, è corretto scrivere "y1 = ain1=3 OR x1" perchè l'espressione inizia con AIN1, che è numerico, ma l'operatore "=" lo confronta con il numero "3" e il risultato è di tipo bit. Il bit risultante viene messo in OR con l'ingresso X1, e il risultato viene assegnato a Y1. L'effetto di questa istruzione è accendere l'uscita 1 se l'ingresso analogico 1 vale 3, o se l'ingresso digitale 1 è ON; l'uscita 1 altrimenti viene spenta. L'espressione "y1 = x1 OR ain1=3" è sbagliata. Essa inizia con un valore bit (x1), che impone di usare la versione OR per i bit, mentre l'operatore OR trova alla sua destra la risorsa AIN1, che è di tipo numerico. Nei casi in cui non si riesce a riordinare l'espressione nel modo corretto, si può fare uso di variabili temporanee; in questo caso si potrebbe scrivere:

```
r1 = ain1=3         ; valore temporaneo
y1 = x1 or r1       ; r1 è uguale ad "ain1=3"
```

Nell'uso di variabili temporanee, occorre prestare attenzione al fatto che vi possono essere task concorrenti che modificano inaspettatamente il valore.

Operatori

Per i tipi bit 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.
#	bit, bit	bit	Disuguaglianza (come XOR)

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 aritmetica.
-	num, num	numero	Sottrazione aritmetica.
*****	num, num	numero	Moltiplicazione.
/	num, num	numero	Divisione intera.

L'operatore NOT

Gli operatori visti finora vogliono due argomenti. L'operatore NOT è speciale perché accetta un singolo operando, alla sua destra, e questo operando deve essere di tipo BIT. Lo scopo di NOT è quello d'invertire il valore alla sua destra: se X1 è ON, “NOT X1” ritorna OFF. Si usa molto nelle espressioni logiche, dove sovente occorre verificare che un certo bit “non sia ON”.

Risorse comuni

Usando soltanto assegnazioni ed espressioni è già possibile creare un ciclo PLC completo. A dire il vero, i normali PLC funzionano esattamente così: ogni uscita, che sia un relé interno o una uscita reale, è il risultato di una espressione logica. Le risorse sono identificate da un nome, come "X" o "Y", e un numero progressivo.

La tabella seguente mostra un elenco di alcune 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. Si raccomanda però di usare DEFINE per dare un nome significativo alle risorse utilizzate in un ciclo.

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	Rappresenta uno degli 8 ingressi digitali, da X1 a X8. Per attivare l'uscita 1 se c'è tensione sul primo o sul secondo ingresso, usare: Y1 = X1 or X2. Lo stato degli ingressi è filtrato ed elaborato. Normalmente, il valore dell'ingresso è ON se sull'ingresso c'è tensione, ma questo può essere cambiato, insieme a diverse opzioni di filtro: vedere il paragrafo specifico, e vedere anche XU, e XD.
XU	8	Rileva i fronti di salita di un ingresso: diventa TRUE quando l'ingresso passa da OFF a ON; deve essere reimpostata a OFF manualmente.
XD	8	Simile a XU; serve per rilevare i fronti di discesa.
Y	8	Rappresenta una delle otto uscite di potenza, da Y1 a Y8. Impostandola a ON (o TRUE), normalmente l'uscita si alza, ma c'è una opzione per invertire le uscite.
XINVERT	8	Imposta l'inversione logica dell'ingresso. Normalmente, un ingresso X è TRUE in presenza di tensione. Se XINVERT per quell'ingresso è TRUE, la logica s'inverte: la risorsa X è TRUE quando sull'ingresso non c'è tensione. Lo scopo è quello di permettere al programmatore di ragionare in logica "diritta".
YINVERT	8	Imposta l'inversione logica dell'uscita, similmente a XINVERT. Se l'uscita Y1 è negata, si porterà tensione su Y1 scrivendo "Y1=OFF".
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.
TIMER	16	Conteggio corrente del timer. Vedere il paragrafo apposito per i timer.
T	16	Stato ON/OFF del timer. Riferirsi al paragrafo specifico per maggiori informazioni.
FB	8	Bit di segnalazione di anomalia sulle uscite di potenza. Vedere il paragrafo specifico per maggiori informazioni.
R	168	Relé interni, a bit, da usare come memorie generiche.
DT	64	Memorie interne, numeriche, a 16 bit. Contengono numeri che vanno da 0 a 65535.
AOUT	2	Uscite analogiche 0-10V. I numeri utilizzabili vanno da 0 (0 volt) a 255 (10 volt). Se s'immette un valore maggiore, esso viene tacitamente ridotto entro l'intervallo attraverso il resto di una divisione intera con 256. Per esempio, il valore 1000 risulta in 1000 mod 256=232.
AIN	2	Ingressi analogici. Ritornano il valore di tensione presente all'ingresso come numero compreso tra 0 (0 volt) e 255 (10 volt).
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 FXCOUTNH si incrementa di 1.
ENCODERL, ENCODERH	1	Posizione dell'encoder (opzionale) collegato agli ingressi veloci FX1 ed FX2. Fare riferimento ala paragrafo relativo.
XCOUNT	2	Conteggio dei fronti di salita sugli ingressi X1 e X2, da 0 a 65535.

Timer TIMER1..TIMER16 (T1..T16)

EZ-Red dispone di 16 timer software, che sono composti da un contatore e da un contatto. Il contatore TIMER può essere letto e scritto normalmente, e si decrementa verso zero automaticamente al ritmo di una unità al

millisecondo. Il contatto relativo, T, è ON se il timer sta contando, e OFF se il timer ha raggiunto lo zero.

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" che significa: accendere l'uscita 1 se FINECORSA è diverso da ON. Naturalmente sarebbe meglio scrivere "Y1 = FINECORSA", ragionando in logica positiva. Ciò è possibile eseguendo un'inversione logica dell'ingresso usando le risorse XINVERT1..XINVERT8.

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 XTRESHOLDUP (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 XTRESHOLDUP, XTRESHOLDDN 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 (meno probabili), 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 XTRESHOLDDN che introduce stabilità (e un ritardo) nel passaggio da ON a OFF.

XTRESHOLDUP e XTRESHOLDDN accettano valori da 0 a 127.

Le risorse XU e XD memorizzano l'avvenuto passaggio di un fronte: XU per i fronti di salita e XD per i fronti di discesa. La loro utilità sta nel fatto che sono in grado di rilevare i fronti anche se il ciclo non sta controllando l'ingresso nel momento preciso in cui avviene il fronte. Se XU1 è TRUE, significa che è stato rilevato un fronte di salita sull'ingresso 1. Il riarmo di questi bit deve essere effettuato in modo manuale, scrivendo "XU1 = OFF".

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. Possono anche essere lette: il valore letto è quello impostato per ultimo. Come per gli ingressi, è possibile invertire logicamente un'uscita ponendo la rispettiva YINVERT a ON.

Feedback (allarme uscite) FB12, FB34, FB56, FB78 (FB1..FB8)

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. I nomi reali di questi bit sono FB12 (non dodici, ma uno-due) per le uscite 1 e 2, poi FB34 (tre-quattro), FB56 ed FB78, ma per semplicità FB12 ha anche i due nomi aggiuntivi FB1 ed FB2, così come FB34 ha FB3 ed FB4 e via dicendo.

I feedback devono essere abilitati usando la risorsa FBENABLE, in cui i primi 4 bit, se a 1, abilitano i rispettivi bit di feedback. All'accensione FBENABLE è 0, quindi i feedback sono disabilitati. Scrivendo FBENABLE=15 si abilitano tutti, mentre pre esempio scrivendo FBENABLE=3 si abilitano solo i primi due (per le prime quattro uscite). Il motivo di questo è che EZ-Red attiva un led di allarme se rileva un allarme sulle uscite, ma questo non è sempre desiderato.

Interfaccia encoder, contatori FXCOUNT e XCOUNT

Gli ingressi veloci FX1 ed FX2, oltre che funzionare da normali ingressi optoisolati, permettono di collegare i due canali di un encoder. In questo caso è possibile leggere la posizione dell'encoder, che conta in avanti e indietro, direttamente come dato numerico dalla risorsa ENCODERL. La "L" sta per "low", e indica che esiste un'altra risorsa ENCODERH dove la "H" sta per "high". Il motivo della divisione in due risorse è che i numeri in EZ-Red sono di 16 bit, consentendo perciò di contare 65536 passi di un encoder. Se la corsa dell'encoder è maggiore di 65536 passi (posizioni), occorre fare uso di ENCODERH che contiene la parte alta del conteggio. Quando un encoder effettua il passo numero 65536, la variabile ENCODERL diventa 0, e la variabile ENCODERH diventa 1; essa contiene il "numero di volte" che ENCODERL si è azzerata. Se si devono effettuare calcoli, occorre considerare le due risorse come è stato appena descritto; se invece si devono solo fare confronti, basta confrontare le due variabili con i rispettivi valori di riferimento. Queste risorse possono anche essere scritte - per esempio azzerate in corrispondenza di un fine corsa.

FXCOUNTL1 ed FXCOUNTH1 (e FXCOUNTL2/FXCOUNTH2) sono contatori a 32 bit associati ai rispettivi ingressi veloci: contano i fronti di salita e possono servire per contare gli impulsi. I 32 bit sono divisi, come per la risorsa ENCODER, in parte bassa e parte alta; anche questi contatori sono scrivibili (per azzerarli).

XCOUNT1 e XCOUNT2 sono contatori di 16 bit associati agli ingressi X1 e X2; il conteggio va da 0 a 65535, ed è possibile scrivere o azzerare il contenuto. Si noti che tutti questi contatori, interfaccia encoder compresa, non dispongono di alcun filtro software; questo è particolarmente vero per gli ingressi X1 e X2: le risorse X1 e X2 sono filtrate ed opzionalmente invertite, ma i contatori associati no.

Controllo di flusso (IF 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 ramo di ladder, anche se potenzialmente più complesso. Le istruzioni IF permettono di eseguire due parti di programma completamente diverse a seconda della situazione, e questo è più vicino al modo di pensare degli esseri umani.

L'istruzione IF ha la forma

IF espressione THEN istruzione_se_vero ELSE istruzione_se_falso

dove la parte ELSE è opzionale. La IF valuta l'espressione, ed esegue l'istruzione dopo THEN se l'espressione è vera; se l'espressione è falsa, e se la parte ELSE esiste, esegue quella.

Si immagini il caso in cui se X1 è ON si vuole accendere Y1, altrimenti Y2. Il testo del programma è "IF x1 THEN y1=on ELSE y2=on".

Un altro esempio è far partire un timer se un pulsante (X1) è premuto: "IF x1 THEN timer1 = 2000".

Sia la parte dopo THEN che quella dopo ELSE possono essere composte da più di una istruzione. Per farlo, basta andare a capo subito dopo THEN o ELSE:

```
IF x1 or x2 or x3 THEN
  y1 = TRUE
  timer1 = 2000
END
```

...e terminare il blocco di istruzioni con END. Se si vuole specificare la parte ELSE dopo un blocco, la parola ELSE deve seguire immediatamente la END:

```
IF x1 or x2 or x3 THEN
  y1 = TRUE
  timer1 = 2000
END ELSE y1 = OFF
```

Anche ELSE può essere seguito da un blocco invece che da una singola istruzione:

```
IF x1 or x2 or x3 THEN
  y1 = TRUE
  timer1 = 2000
END ELSE
  y1 = OFF
  DT3 = DT3+1
END
```

Un blocco di istruzioni può contenere quante istruzioni si vogliono, di qualsiasi tipo incluso altre IF, ma non dichiarazioni DEFINE o di etichette (difatti queste due 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.

L'istruzione GOTO potrebbe essere superflua, facendo uso di una serie di IF ben costruite. Tuttavia, in determinati casi, l'uso di GOTO è più comodo e leggibile. 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:
  ...
```

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:
  ...
```

ATTENZIONE: 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.

Istruzioni e Risorse speciali

Controllo dei 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 un task quando è sospeso occorre usare l'istruzione WAKEUP seguita dal numero di task. Il numero di task deve essere espresso con una costante, o un identificatore dichiarato con una DEFINE riferita a una costante:

```
WAKEUP 2     ; attiva il task 2
             ; se già attivo, l'istruzione è ignorata
```

L'istruzione RESTART è molto simile a WAKEUP, ma 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
```

Un task può essere anche sospeso con l'istruzione SUSPEND; il task indicato viene bloccato nel punto in cui si trova, e al momento del WAKEUP riprende dallo stesso punto. Per questo motivo, di solito è meglio e più facile sospendere un task dall'interno del task medesimo, piuttosto che da un altro task: in altre parole, è meglio che un task sospenda sè stesso, in una posizione conosciuta:

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

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

L'istruzione SUSPEND è seguita dal numero del task da sospendere; se omesso, viene sospeso il task corrente. Per esempio:

```
Task2:
  Y1 = ON
  wait 500
  Y1 = OFF
  suspend
```

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 deve essere specificato con una costante; può essere zero (oppure omesso): in tal caso l'attesa è di 1 millisecondo.

Che si 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 da NOT; si usa NOT per indicare che si deve attendere che il bit diventi FALSE:

```
WAIT 1000           ; pausa di un secondo
WAIT X1 X2          ; attende che X1 o X2 siano ON
WAIT 100 X1         ; attende X1 on, per 100 ms
WAIT NOT T1         ; attende che il timer T1 cada
WAIT 1000 X1 X2 NOT T1 NOT T2
  ; attende, al massimo un secondo, che X1 o X2
  ; diventino ON oppure che T1 o T2 espirino
```

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

```
WAIT
WAIT 0
WAIT 1
  ; queste tre istruzioni ritardano di 1 ms

WAIT 0 X1
WAIT X1
  ; queste due istruzioni attendono, anche per sempre,
  ; che X1 diventi true
```

Precisazione sul tempo d'attesa: se si specifica ***n*** come attesa, 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:

```
Task2:
  y2=ON
  WAIT 1
  y2=OFF
  WAIT 1
  goto Task2
```

e il motivo è semplice: le istruzioni "y2=on", "y2=off" e "goto Task2" richiedono tempo per essere eseguite; questo tempo verrebbe sommato ai due millisecondi richiesti dalle WAIT, e quindi 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 si sincronizza su un tick interno, che gira libero a 1 Khz; l'effetto finale 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.

Risorse speciali di tipo BYTE

Diversi aspetti degli I/O possono essere gestiti in modo individuale (bit per bit), ma talvolta, specie in fase di

iniziazione, può essere comoda una maggiore concisione usando alcune risorse speciali elencate nella seguente tabella:

Nome	Descrizione
XBYTE	Legge gli otto ingressi X1..X8 in un colpo solo; ogni bit corrisponde a un ingresso. Quindi, "XBYTE and 1 # 0" è equivalente a X1, mentre "XBYTE and 128 # 0" corrisponde a X8.
YBYTE	Come XBYTE, ma per le uscite. Per accendere tutte le uscite, basta usare "YBYTE = 255" e, per spegnerle, "XBYTE = 0".
XUBYTE	Come XBYTE, ma contiene i fronti di salita di ogni ingresso.
XDBYTE	Come XUBYTE, ma per i fronti di discesa.
XINVBYTE	Contiene i flag di inversione degli ingressi X1..X8.
YINVBYTE	Contiene i flag di inversione delle uscite di potenza Y1..Y8.
FBENABLE	Già descritto in precedenza, contiene un bit ogni due uscite (quindi 4 bit). I rimanenti quattro bit superiori non sono usati.

Risorse speciali di tipo BIT

La seguente tabella riporta nome e spiegazione di alcuni bit speciali, che possono essere letti o scritti per scopi particolari:

Nome	Descrizione
REPORTBACK	Indica o imposta il "report back". Quando il modo è attivo, un cambiamento sulle uscite provoca una trasmissione USB al computer, che indica il nuovo stato degli ingressi.
WDTFIRED	Indica che il Watch-Dog è intervenuto. Può essere anche scritto: per resettarlo o, se messo a ON, per scatenare lo stesso effetto di un intervento del watch-dog.
SENDTOPC	Ponendolo a ON, forza l'invio di un report al PC, come se REPORTBACK fosse attivo e lo stato degli ingressi fosse cambiato. Dopo la trasmissione (qualche millisecondo), ritorna automaticamente a OFF.
CYCLERUN	Se il ciclo è attivo è sempre ON. Portandolo a OFF si ferma l'esecuzione del ciclo in modo irreversibile (è necessario riavviare il ciclo usando il computer).
WDTSTOPSCYCLE	Se ON (default), ferma il ciclo quando il watch-dog interviene.
DISABLEUSB	DA USARE CON MOLTA ATTENZIONE Se attivato, la comunicazione tramite USB viene impedita. Alzare questo bit da ciclo fa sì che un dispositivo EZ-Red non possa più essere riprogrammato, e neppure utilizzato insieme al computer. E' possibile tuttavia alzare il bit da ciclo solo dopo un certo ritardo, o al verificarsi di determinate condizioni, in modo da lasciare aperta una strada per resettarlo. Oppure è possibile, sempre all'interno del ciclo, prevedere un insieme di condizioni o una manovra particolare per disattivare il flag e consentire l'utilizzo del computer.
SYNC_IO	Inibisce l'aggiornamento asincrono degli I/O. Gli I/O digitali, quelli analogici e gli stati dei timer non vengono più aggiornati ogni millisecondo, ma solo su richiesta esplicita con il comando UPDATEIO. Serve a rendere EZ-Red più compatibile con i PLC classici.

Watch-dog

Nel caso in cui il ciclo preveda l'interazione con un personal computer, via USB, e si pensa che il computer, per un motivo qualunque, possa bloccarsi portando a conseguenze indesiderabili, è possibile abilitare il watch-dog interno di EZ-Red. Quando il watch-dog è attivo, EZ-Red attende almeno una trasmissione USB dal computer ogni N millisecondi; se trascorsi questi N millisecondi nessuna comunicazione è arrivata, si assume che il computer si è bloccato e il watch-dog interviene.

Quando il watch-dog interviene, le uscite vengono portate a uno stato preciso, detto di emergenza, in modo che si fermino eventuali motori in movimento, forni che scaldano, e simili potenziali pericoli. In aggiunta, il ciclo viene fermato (opzione escludibile).

Per usare il watch-dog occorre impostare le seguenti risorse (tutte opzionali tranne l'ultima):

1. WDTOUTS (byte). Contiene la configurazione da assegnare alle uscite quando il watch-dog interviene. Il suo formato è simile a YBYTE, e il valore iniziale è 192 (le uscite da 1 a 6 basse, la 7 e la 8 alte). Le due uscite alte possono servire per segnalare un allarme.
2. WDTAOUT1 (byte). Contiene il valore da assegnare all'uscita analogica 1, quando il watch-dog interviene. Il valore iniziale è 0.
3. WDTAOUT2 (byte). Come WDTAOUT1, per l'uscita analogica 2.
4. WDTSTOPSCYCLE (bit). Se TRUE (default), l'intervento del watch-dog ferma anche il ciclo. Impostare a FALSE o OFF se si vuole gestire la sitauzione da ciclo.
5. WDTTIME (word). Contiene, in millesimi di secondo, il tempo di intervento.

Interazione Watch-dog e ciclo PLC

In condizioni normali, WDTSTOPSCYCLE è ON e, quando il watch-dog interviene, le uscite vengono impostate ai valori di emergenza 100 volte al secondo. Quando WDTSTOPSCYCLE è FALSE il ciclo non viene fermato: è responsabilità dello stesso rilevare lo stato di emergenza e intraprendere le azioni opportune – le uscite NON vengono modificate.

Se la condizione di anomalia deve essere gestita da ciclo, si può fare in due modi – in entrambi i modi occorre tenere sotto controllo la variabile WDTFIRED. Il primo modo è quello di controllare la variabile ciclicamente, con una semplice IF-THEN; però vi può essere una latenza più o meno lunga. Il secondo modo usa un task apposito destinato al watch-dog; così facendo la latenza è minima, perché il task attende con WAIT e si sveglia appena il watch-dog interviene:

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

  ; se arriva qui, è perché il WDT è intervenuto
  SUSPEND 1
  SUSPEND 2

  ...  ; altre azioni
```

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

Esempi di programmi

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

Pressa con controllo di sicurezza

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.

```
; Controllo pressa

define pressaY3
define puls1 x1
define puls2 x2

XTHRESHOLDUP1 = 100
XTHRESHOLDUP2 = 100
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
```

L'impostazione di XTHRESHOLDxx assicura che non ci siano rimbalzi nei pulsanti.

Braccio in spinta con pulsante e 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; appena il comando cessa, la slitta deve tornare indietro. La slitta si ferma anche sul fine corsa avanti; il comando non deve durare più di dieci secondi.

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 programma ladder normale. La prima parte del programma definisce in modo mnemonico tempi e risorse utilizzati:

```
; Comando slitta

define tempo_valido 10000 ; tempo di validità del comando

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

define com_timer    timer1
define com_valido   t1     ; 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 timer di validità parte quando il pulsante d'avvio è
; rilasciato in corrispondenza del fine corsa indietro
if not com_avanti and not ok_indietro then
  com_timer = tempo_valido
end

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

; andare indietro se il pulsante è rilasciato o
; il comando è scaduto – fino al fine corsa indietro
mot_indietro = not com_avanti or not com_valido and ok_indietro
```

Il programma è parzialmente procedurale perché usa una istruzione IF. Questo è dovuto al fatto che per attivare il timer occorre usare un'istruzione che non è compatibile con i dati di tipo bit.

Il programma è composto principalmente da tre istruzioni: la IF, la “MOT_AVANTI=...” e la “MOT_INDIETRO=...”. Dopo quest'ultima istruzione il programma riparte dalla IF; tutte le DEFINE all'inizio del testo non fanno effettivamente parte del ciclo – esistono solo durante la compilazione.

Questo programma presenta un lieve difetto: se il timer COM_VALIDO scade subito dopo l'istruzione MOT_AVANTI=, ma prima di MOT_INDIETRO=, succede che il ciclo attiva contemporaneamente il comando motore avanti e quello motore indietro (per alcuni microsecondi). Ciò può succedere, perché i timer corrono da soli, senza nessun legame con l'esecuzione del ciclo; il programma esegue la lettura del timer in due istruzioni separate, assumendo però che il valore non sia cambiato. Il problema andrebbe risolto leggendo il valore del timer una volta sola; si dovrebbe usare una variabile temporanea, che non può modificarsi da sola, e fare poi riferimento alla variabile invece che direttamente ai timer:

```
; il timer di validità parte quando il pulsante d'avvio è
; rilasciato in corrispondenza del fine corsa indietro
if not com_avanti and not ok_indietro then
  com_timer = tempo_valido
end

; andare avanti se pulsante premuto e comando valido
; e se il fine corsa non è ingaggiato (ok_avanti)
; leggere il timer in modo “atomico”
R1 = com_valido
mot_avanti = com_avanti and R1 and ok_avanti

; andare indietro se il pulsante è rilasciato o
; il comando è scaduto – fino al fine corsa indietro
mot_indietro = not com_avanti or not R1 and ok_indietro
```

Anche così, si possono verificare brevi sovrapposizioni dei due comandi avanti e indietro. Dato che l'aggiornamento degli I/O è normalmente asincrono, può succedere che esso avvenga tra MOT_AVANTI= e MOT_INDIETRO=, quando MOT_INDIETRO è ancora ON dal giro precedente, e MOT_AVANTI diventa ON nel giro corrente. In questo caso, la sovrapposizione indesiderata dura un millisecondo.

La soluzione di questi problemi è: impostare il bit SYNC_IO. Se il bit è impostato, l'aggiornamento non avviene più ogni millisecondo in modo asincrono, il che può comportare i problemi descritti, ma solo quando l'apposita istruzione UPDATEIO viene eseguita. Questo bit può essere impostato in fase di iniziazione, o continuamente durante il ciclo, oppure temporaneamente quando serve. Questo programma programma potrebbe essere scritto così:

```
SYNC_IO = ON

if not com_avanti and not ok_indietro then
  com_timer = tempo_valido
end

mot_avanti = com_avanti and com_valido and ok_avanti
mot_indietro = not com_avanti or not com_valido and ok_indietro
UPDATEIO
```

oppure nel seguente modo, dove il bit viene usato solo per impedire che il timer o l'aggiornamento asincrono degli I/O accadano in mezzo alle due istruzioni del comando del motore:

```

if not com_avanti and not ok_indietro then
  com_timer = tempo_valido
end

SYNC_IO = ON        ; disabilitare temporaneamente
mot_avanti = com_avanti and com_valido and ok_avanti
mot_indietro = not com_avanti or not com_valido and ok_indietro
SYNC_IO = OFF       ; e riabilitare subito dopo
```

L'orientamento di fondo di EZ-Red prevede uno stile di programmazione procedurale, che mal si sposa con l'I/O sincrono, specialmente nei casi di lunghe pause. Per questo motivo, quando si pongono problemi come quelli descritto, si raccomanda di risolverli disabilitando solo temporaneamente l'I/O asincrono.

Braccio in spinta con pulsante e 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 le stesse definizioni del programma precedente:

```
; Comando slitta

define tempo_valido 10000 ; tempo di validità del comando

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

define com_timer    timer1
define com_valido   t1    ; timer di validità comando
  ; indica che il comando del pulsante non ha superato
  ; i 10 secondi
```

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 com_avanti=false then com_timer = tempo_valido

  ; verificare comando
  if com_avanti and com_valido then goto avanti
  goto riposo

Avanti:
  mot_indietro = OFF
  mot_avanti = ON
  ; attendere: scadenza, o fine corsa, o fine comando
  wait not ok_avanti not com_avanti not com_valido
  if not com_valido then goto riposo   ; tempo scaduto
  if not com_avanti then goto riposo   ; rilasciato

  ; arrivato sul fine-corsa, spegnere motore
  mot_avanti = OFF
  wait not com_avanti not com_valido

  ; il ciclo riparte automaticamente
```

Il programma è più lungo, ma analizza in maggior dettaglio tutte le singole fasi; inoltre non ci sono problemi di asincronicità degli I/O perché la logica è procedurale.

Collaudo ingressi/uscite

Questo esempio mostra bene che in certi casi la logica procedurale è più facile di quella funzionale; lo scopo del

programma è verificare che ingressi e uscite del modulo funzionino correttamente - un collaudo basilare (e incompleto). Assumiamo che ogni ingresso del modulo sia collegato con la rispettiva uscita (X1 con Y1 e così via); anche la parte analogica: AIN1 con AOUT1 e AIN2 con AOUT2. Gli ingressi veloci FX1 ed FX2 sono collegati insieme con Y8.

Il programma spegne innanzitutto le uscite di potenza e, dopo una pausa di assestamento, controlla che tutti gli ingressi digitali siano a OFF. Questo può essere fatto in due modi:

```
if x1 or x2 or x3 or x4 or x5 or x6 or x7 or x8 then goto error
if xbyte # 0 then goto error
```

Le due righe di programma fanno esattamente la stessa cosa; naturalmente la seconda è più rapida e concisa perché analizza gli 8 ingressi in un colpo solo. Se si verifica una discrepanza, cioé un ingresso è ON (ma non dovrebbe esserlo, per come è fatto il cablaggio), si salta alla parte di gestione dell'errore.

Gli ingressi FX1 ed FX2 sono collegati a Y8, e controllati in modo analogo.

In un secondo passo, lo stesso viene fatto ponendo le uscite a ON, e controllando che tutti gli ingressi vadano a ON:

```
r1 = x1 and x2 and x3 and x4 and x5 and x6 and x7 and x8
if r1 # true then goto error
if not fx1 then goto error
if not fx2 then goto error
```

L' uso della variabile temporanea R1 serve a evitare una noiosa ripetizione di "not x1 or not x2...". Anche in questo caso si sarebbe potuto usare la variable XBYTE per leggere il gruppo di ingressi.

Per il collaudo di ingressi e uscite analogiche si segue un metodo simile.

Se non ci sono discrepanze, il programma prosegue all'etichetta **collaudo_ok**, dove il programmatore si diverte a fare un gioco luminoso.

```
collaudo_ok:
  ybyte = 1

scorre:
  wait 50
  ybyte = ybyte*2
  if ybyte=0 then goto collaudo_ok
  goto scorre
```

Con "YBYTE=1" si accende sono l'uscita 1. Il ciclo poi fa ruotare il bit (moltiplicare per due equivale a far scorrere i bit a destra), accendendo così le varie uscite in sequenza. Quando è accesa l'uscita 8, una moltiplicazione ulteriore porta al valore 256, che in binario risulta in 1.0000.0000; ma quando questo valore viene scritto dentro YBYTE, solo gli 8 bit più a destra possono essere scritti. Il risultato è che YBYTE diventa 0, e tutte le uscite si spengono. Subito dopo, però, la "if ybyte=0..." intercetta la condizione, e fa ripartire il ciclo dall'inizio.

Il listato completo è il seguente:

```
; Collaudo ingressi/uscite

; verificare che gli ingressi digitali vadano in OFF
ybyte=0
wait 1000
if x1 or x2 or x3 or x4 or x5 or x6 or x7 or x8 then goto error
if xbyte # 0 then goto error
if fx1 or fx2 then goto error

; verificare che tutto sia acceso
ybyte=255
wait 1000
r1 = x1 and x2 and x3 and x4 and x5 and x6 and x7 and x8
if r1 # true then goto error
if not fx1 then goto error
if not fx2 then goto error

; verificare ain1
aout1 = 16
wait 100
if ain1 < 8 then goto error
if ain1 > 24 then goto error
aout1 = 192
wait 100
if ain1 < 186 then goto error
if ain1 > 200 then goto error

; verificare ain2
aout2 = 16
wait 100
if ain2 < 8 then goto error
if ain2 > 24 then goto error
aout2 = 192
wait 100
if ain2 < 186 then goto error
if ain2 > 200 then goto error


; tutto funziona
collaudo_ok:
  ybyte = 1

scorre:
  aout1 = aout1+10
  aout2 = aout2-5
  wait 50
  ybyte = ybyte*2
  if ybyte=0 then goto collaudo_ok
  goto scorre

error:
  ; qui segnala errore
  ybyte=255
  wait 200
  ybyte=0
  wait 200
  goto error
```

Controllo motore con encoder

In questa applicazione si supponga di 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 e azzera la quota.

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 indietro
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
  xinvert2 = true   ; il fine corsa è invertito (0=fine corsa)
  speed=10
  mot_indietro = OFF
  mot_avanti = ON;
  wait 5000 not finecorsa

  ; o timeout, o uscito dal fine corsa
  mot_avanti = OFF
  if finecorsa then goto errore

  ; azzerare sul micro
  mot_indietro = ON
  wait 15000 not finecorsa
  mot_indietro = OFF
  if finecorsa then goto Ciclo

Errore:
; ========== errore
  wdtfired = ON;
  goto errore
```

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
  encoderl=0
  ; attendere pulsante start
  wait start

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

  ; tornare indietro - se motore non fermo, arrestare dolcemente
 attesa:
  ; diminuisce la velocità fino a 10
  if speed > 10 then
    speed = speed-1
    wait 5
    goto attesa
  end
  wait 100

  mot_avanti=off
  wait 100
  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 ZERO
  ; se vicino all'arrivo, diminuisce la velocità
  ; se no, la aumenta
  dt1 = encoderL
  if mot_avanti then dt1 = quota - dt1
  if dt1 < 900 then
    ; vicini all'arrivo - rallentare
    speed = dt1 / 5
    if dt1<5 then
       ; se motore avanti, fermare
      if mot_avanti then speed=0
    end
  end else
    ; se possibile, aumentare
    if speed<200 then
       speed=speed+1
       wait 5
    end
  end
```