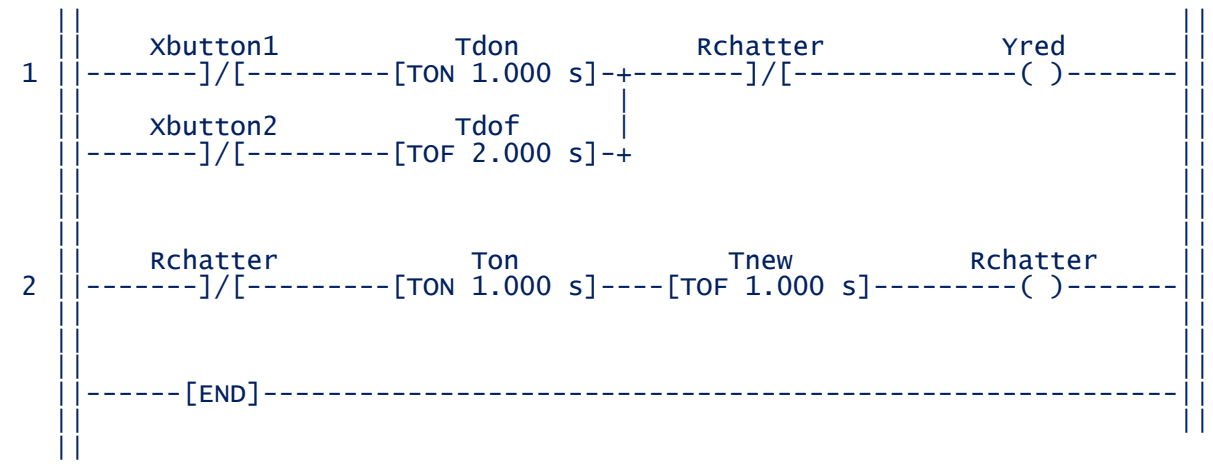
Manuale LD MICRO

INTRODUZIONE

==========

LDmicro genera codice nativo per alcuni microcontrollori Microchip PIC16 e Atmel AVR . Normalmente il software per questi microcontrollori è scritto in un linguaggio di programmazione come assembler, C, o BASIC. Un programma in uno di queste linguaggi dispone di un elenco di istruzioni. Queste linguaggi sono potenti e ben adatti per l'architettura del processore, che esegue internamente una lista di istruzioni. Il PLC, d'altra parte, sono spesso programmati in `logica ladder. 'Un semplice programma potrebbe essere simile a questo:



(TON è un turn-on delay, TOF è un turn-off Il ritardo

-] [- dichiarazioni. sono ingressi, che si comportano un po 'come i contatti di un relè.

Il - () - Dichiarazioni sono uscite, che si comportano un po 'come la bobina di un relè.

Molte buone referenze per la logica ladder sono disponibili su Internet e altrove, i dettagli specifici a questa esecuzione sono indicati di seguito). Un certo numero di differenze sono evidenti:

* Il programma è in formato grafico, non come elenco testuale di istruzioni. Molte persone inizialmente trovare questo più facile da capire.

* Al livello più elementare, programmi di apparire come schemi elettrici, con contatti relè (ingressi) e delle bobine (uscite). Questo è intuitivo programmatori con conoscenze di teoria dei circuiti elettrici.

* Il compilatore logica ladder si prende cura di ciò che viene calcolato dove. Non è necessario scrivere codice per determinare quando le uscite devono avere ricalcolati sulla base di una variazione degli ingressi o evento timer, e non c'è bisogno di specificare l'ordine in cui questi calcoli devono avvenire: i programmi di PLC farlo per voi.

LDmicro compila la logica ladder per PIC16 o codice AVR. l seguentii processori sono supportati:

* PIC16F877 * PIC16F628 * PIC16F876 * PIC16F88 * PIC16F819 * PIC16F887 * PIC16F886 * ATmega128 * ATMEGA64 * ATMEGA162 * *ATmega32* * ATmega16 * ATmega8

Sarebbe facile per supportare più AVR o chip PIC16, ma non ho un modo per testarli. Se avete bisogno di uno, in particolare, poi in contatto con me e Vedrò cosa posso fare.

Utilizzando LDmicro, è possibile disegnare un diagramma a relè per il vostro programma. È possibile simulare la logica in tempo reale sul computer. Poi, quando si è convinti che sia corretta è possibile assegnare pin del microcontrollore al ingressi e le uscite del programma. Una volta assegnati i pin , è possibile compilare PIC AVR o codice per il programma. L'uscita è un compilatore. Eseguono file che è possibile programmare nel vostro microcontrollore utilizzando qualsiasi programmatore PIC / AVR.

LDmicro è progettato per essere in qualche modo simile al più commerciale PLC programmazione di sistemi. Ci sono alcune eccezioni, e un sacco di cose non sono standard in ogni modo di settore. Leggere attentamente la descrizione di ciascuna istruzione, anche se sembra familiare. Questo documento assume

conoscenza di base della logica ladder e della struttura del software PLC (Il ciclo di esecuzione: leggere gli ingressi, calcolare, scrivere uscite).

Ulteriori obiettivi

==========

È anche possibile generare codice C ANSI. È possibile utilizzare questo con qualsiasi processore per il quale si dispone di un compilatore C, ma sei responsabile di fornendo il runtime. Ciò significa che LDmicro genera solo fonte per un PlcCycle funzione (). L'utente è responsabile per chiamare PlcCycle ogni tempo di ciclo, e sono responsabili dell'attuazione tutti gli I / O (Lettura / scrittura ingresso digitale, ecc), le funzioni che il PlcCycle () chiama. Vedere i commenti nel sorgente generato per ulteriori dettagli.

Infine, LDmicro può generare bytecode indipendente dal processore di macchina virtuale progettato per eseguire codice della logica ladder. Ho fornito un implementazione di esempio dell'interprete / VM, scritto in C abbastanza portatile. Questo obiettivo funziona per quasi tutte le piattaforme, fino a quando si è in grado di fornire il VM proprio. Questo potrebbe essere utile per le applicazioni in cui si desidera usare la logica ladder come `linguaggio di scripting 'di personalizzare una più grande programma. Vedi i commenti l'interprete del campione per i dettagli.

Opzioni della riga comandi

==========

ldmicro.exe viene in genere eseguito con le opzioni della riga di comando senza. Ciò significa che si può solo creare un collegamento al programma, o salvarlo sul desktop e fare doppio clic l'icona quando si desidera eseguire, e quindi si può fare tutto da all'interno della GUI. Se LDmicro viene passato un nome di file unico sulla riga di comando (Ad esempio, `ldmicro.exe asd.ld '), poi LDmicro tenterà di aprire` asd.ld', se esiste. Un errore viene prodotto se `asd.ld 'non esiste. Questo significa che è possibile associare con ldmicro.exe. ld file, in modo che venga eseguito automaticamente quando si fa doppio clic su un file. ld. Se LDmicro viene passato gli argomenti della riga di comando in forma `Ldmicro.exe / c src.ld dest.hex ', poi si cerca di compilare` src.ld', e salvare l'output come `dest.hex '. Esce LDmicro dopo la compilazione, se la compilazione ha avuto successo o meno. Tutti i messaggi vengono stampati alla console. Questa modalità è utile solo quando si esegue LDmicro da riga di comando.

Se si esegue LDmicro senza argomenti, allora inizia con un vuoto programma. Se si esegue LDmicro con il nome di un programma ladder (xxx.ld) sulla riga di comando, allora cercherà di caricare il programma all'avvio. LDmicro utilizza un proprio formato interno per il programma, ma non è possibile importare logica da qualsiasi altro strumento.

Se non è stato caricato un programma esistente allora si sarà dato un programma con un ramo vuoto. Si potrebbe aggiungere un'istruzione ad esso, ad esempio è possibile aggiungere una serie di contatti istruzione -> Contatti Inserisci) di nome`Xnew '. `X 'significa che i contatti saranno legati a un pin di ingresso sul microcontrollore. Si potrebbe assegnare uno spillo in un secondo momento, dopo aver scelto un microcontrollore e rinominare i contatti. La prima lettera di un nome indica il tipo di oggetto si tratta. Per esempio:

* XName - legato ad un pin di ingresso sul microcontrollore

* Yname - legata ad un pin di uscita del microcontrollore

* RNAME - `relè interno ': un po' di memoria

* TNome - un timer, accensione ritardo, ritardo di spegnimento, o ritentivo

* Cname - un contatore, o count-up o count-down

* Aname - un intero letto da un convertitore A / D

* Nome - una general-purpose (intero) variabile

Scegliere il resto del nome in modo che venga descritta la funzione dell'oggetto, e in modo che sia univoco all'interno del programma. Lo stesso nome si riferisce sempre allo stesso oggetto all'interno del programma. Ad esempio, sarebbe un errore di avere un turn-on delay (TON) chiamato `Tdelay 'e un turn-off ritardo (TOF)

chiamato `Tdelay 'nello stesso programma, dal momento che ogni contatore ha bisogno di una propria memoria. D'altra parte, sarebbe corretto avere un timer ritentivo (RTO) chiamato `Tdelay 'e un'istruzione di reset (RES) associato `Tdelay ', dal momento che questo caso si desidera che entrambi i istruzioni per lavorare con lo stesso timer.

I nomi delle variabili può essere costituito da lettere, numeri e caratteri di sottolineatura (_). Un nome di variabile non deve iniziare con un numero. I nomi delle variabili sono tra maiuscole e minuscole.

Le istruzioni generali variabili (MOV, ADD, EQU, ecc) può lavorare su variabili con qualsiasi nome. Ciò significa che possono accedere timer e accumulatori contatore. Ciò può rivelarsi utile, ad esempio, è

potrebbe verificare se il conteggio di un timer è in un determinato intervallo.

Le variabili sono sempre 16 bit interi. Ciò significa che possono andare -32.768-32.767. Le variabili sono sempre trattati come firma. È possibile specificare valori letterali come normali numeri decimali (0, 1234, -56). È anche possibile specificare i valori dei caratteri ASCII ('A', 'z') mettendo il personaggio in apici. È

possibile utilizzare un codice di carattere ASCII nella maggior parte dei luoghi che è possibile utilizzare un numero decimale.

Nella parte inferiore dello schermo viene visualizzato un elenco di tutti gli oggetti in il programma. Questo elenco viene generato automaticamente dal programma; non vi è alcuna necessità di mantenere aggiornata a mano. La maggior parte degli oggetti non bisogno di alcuna configurazione. XName `',` Yname', e `AName 'oggetti devono essere assegnato ad un perno sul microcontrollore, tuttavia. In primo luogo scegliere quale microcontrollore si sta utilizzando (Impostazioni - Microcontrollori>). Quindi assegnare il pin I / O con un doppio clic su di essi nell'elenco.

È possibile modificare il programma di inserimento o eliminazione di istruzioni. Il cursore lampeggia sul display del programma per indicare il selezionato istruzione e il punto di inserimento corrente. Se non lampeggia poi <Tab> premere o fare clic su un'istruzione. Ora è possibile eliminare l'attuale istruzioni, oppure è possibile inserire una nuova istruzione a destra oa sinistra (In serie) o al di sopra o al di sotto (in parallelo) selezionato istruzioni. Alcune operazioni non sono ammessi. Per esempio, senza le istruzioni sono autorizzati a destra di una bobina.

Il programma inizia con un solo gradino. È possibile aggiungere più pioli selezionando Inserire Rung Prima / Dopo nel menu Logic. Si potrebbe ottenere lo stesso effetto ponendo molti sottocircuiti complicati in parallelo all'interno di un gradino, ma è più evidente per utilizzare più rami. Una volta scritto un programma, è possibile eseguirne il test in simulazione, e poi è possibile compilare un file HEX per il microcontrollore.

SIMULAZIONE

==========

Per entrare in modalità di simulazione, selezionare Simula -> Modalità Simulazione o premere <Ctrl+M>. Il programma viene visualizzato in modo diverso in modalità di simulazione. C'è non più un cursore. Le istruzioni che vengono eccitati presentarsi luminoso rosso, le istruzioni che non vengono visualizzati in grigio. Premere la barra spaziatrice per eseguire un ciclo PLC. A ciclo continuo e in tempo reale, scegliere Simulate -> Avvia simulazione in tempo reale, o <Ctrl+R> premere. La visualizzazione di il programma sarà aggiornato in tempo reale come cambia lo stato del programma.

È possibile impostare lo stato degli ingressi al programma facendo doppio clic nella lista nella parte inferiore dello schermo, oppure facendo doppio clic su un `'Istruzione XName contatti nel programma. Se si modifica lo stato di un pin di ingresso quindi che il cambiamento non si riflette nel modo in cui il programma viene visualizzato fino a quando il ciclo del PLC, questo avverrà automaticamente se si esegue una simulazione in tempo reale, o quando si preme la barra spaziatrice.

La compilazione in codice nativo

=======================

In definitiva il punto è quello di generare un file. Hex file che è possibile programmare nel microcontrollore. In primo luogo è necessario selezionare il numero di parte del microcontrollore, sotto Impostazioni -> menu microcontrollore. Poi si necessario assegnare un pin I / O per ogni `XName 'o` Yname' oggetto. Per fare ciò, doppio clic sul nome dell'oggetto nella lista nella parte inferiore dello schermo. Una finestra di dialogo pop-up in cui è possibile scegliere un pin non allocato da un elenco.

Poi si deve scegliere il tempo di ciclo che si correrà con, ed è necessario dire al compilatore quale velocità di

clock del micro sarà in esecuzione. Queste sono fissati in base alle Impostazioni -> Parametri MCU ... menu. in generale si non dovrebbe essere necessario modificare il tempo di ciclo, 10 ms è un buon prezzo per la maggior parte applicazioni. Digitare la frequenza del cristallo che si intende utilizzare con il icrocontrollore (o il risonatore di ceramica, ecc) e fare clic su ok.

Ora è possibile generare il codice dal programma. Scegliere Compila -> Compila, o Compila -> Compila con come ... se è già compilato questo programma e si desidera specificare un diverso nome del file di output. Se non ci sono gli errori di allora LDmicro genererà un file Intel IHEX pronto per programmare nel chip.

Utilizzare qualsiasi software di programmazione e hardware è necessario caricare il tipo dimicrocontrollore. Ricordate di impostare i bit di configurazione (Fusibili)! Per i processori PIC16, i bit di configurazione sono inclusi nella hex, e il software di programmazione sarà più lì automaticamente. Per i processori AVR è necessario impostare i bit di configurazione a mano.

ISTRUZIONI DI RIFERIMENTO

==========

> **Contatto NO XName RNAME Yname**

----] [--------] [--------] [----

Se il segnale in ingresso al istruzione è falsa, allora l'uscita del segnale è falso. Se il segnale in ingresso al istruzione è vera, quindi il segnale di uscita è vera se e solo se il pin di ingresso dato, relè di pin di uscita, o interno è vero, altrimenti è falsa. Questo istruzione può esaminare lo stato di un pin di ingresso, un pin di uscita, o di un relè interno.

> **CONTATTO, normalmente chiusa XName Yname RNAME**

----] / [--------] / [--------] / [----

Se il segnale in ingresso al istruzione è falsa, allora l'uscita segnale è falso. Se il segnale in ingresso al istruzione è vera, quindi il segnale di uscita è vera se e solo se il pin di ingresso dato, relè di pin di uscita, o interno è falso, altrimenti è falsa. Questo istruzione può esaminare lo stato di un pin di ingresso, un pin di uscita, o di un relè interno. Questo è l'opposto di un contatto normalmente aperto.

> COIL, Yname RNAME NORMALE

---- () -------- () ----

Se il segnale in ingresso al istruzione è falsa, allora il dato relè interno o pin di uscita viene azzerato falso. Se il segnale che in questa istruzione è vera, allora il relè interno o data di uscita pin è impostata su true. Non ha senso assegnare una variabile di ingresso ad un coil. Questa istruzione deve essere l'istruzione più a destra nel suo ramo.

> COIL, Yname RNAME negati

---- (/) -------- (/) ----

Se il segnale in ingresso al istruzione è vera, allora la data relè interno o pin di uscita viene azzerato falso. Se il segnale che in questa istruzione è falsa, allora il relè dato interno o pin di uscita è impostata su true. Non ha senso assegnare un ingresso variabile di una bobina. Questo è l'opposto di una bobina normale.

Questo istruzione deve essere l'istruzione più a destra nel suo ramo.

> COIL, SET-SOLO Yname RNAME

---- (S) -------- (S) ----

Se il segnale in ingresso al istruzione è vera, allora la data relè interno o pin di uscita è impostata su true. Altrimenti l'interno relè o pin di uscita non è stato cambiato. Questa istruzione non può che cambiare lo stato di una bobina da false a true, quindi è tipicamente utilizzato in combinazione con un reset sola bobina. Questa istruzione deve essere è l'istruzione più a destra nel suo ramo.

> **COIL, RESET-SOLO Yname RNAME**

---- (R) -------- (R) ----

Se il segnale in ingresso al istruzione è vera, allora la data relè interno o pin di uscita viene azzerato falso. Altrimenti l' relè interno o perno stato delle uscite non viene modificato. Questa istruzione istruzione può modificare solo lo stato di una bobina da true a false, quindi è tipicamente utilizzato in combinazione con un insieme di sola bobina. Questo istruzione deve essere l'istruzione più a destra nel suo ramo.

> **TURN-ON DELAY Tdon**

- [TON 1.000 s] -

Quando il segnale in ingresso al di istruzioni va da false a true, il segnale di uscita rimane false per 1.000 prima di andare vero. Quando l' segnale in ingresso al di istruzioni va da true a false, l'uscita segnale diventa falsa immediatamente. Il timer viene azzerato ogni volta che l'ingresso diventa falsa, l'ingresso deve rimanere fedele per 1000 millisecondi consecutivi prima che l'uscita andrà vera. Il ritardo è configurabile.

I `tNome 'conta variabile fino da zero in unità di tempi di scansione. Il Istruzione TON restituisce true quando la variabile contatore è maggiore o uguale al ritardo determinato. È possibile manipolare la variabile contatore altrove, per esempio con un'istruzione MOV.

> TURN-OFF DELAY Tdoff

- [TOF 1,000 s] -

Quando il segnale in ingresso al di istruzioni va da true a false, il segnale di uscita rimane fedele per 1.000 s prima di andare falso. Quando il segnale in ingresso al di istruzioni va da false a true, il segnale di uscita diventa vera immediatamente. Il timer viene azzerato ogni volta che l'ingresso diventa falso, l'ingresso deve rimanere falso per 1000 millisecondi consecutivi prima l'uscita andrà falsa. Il ritardo è configurabile.

I `**tNome** 'conta variabile fino da zero in unità di tempi di scansione. Il Istruzione TON restituisce true quando la variabile contatore è maggiore o uguale al ritardo determinato. È possibile manipolare la variabile contatore altrove, per esempio con un'istruzione MOV.

> **TIMER RITENTIVO TRTO**

- [RTO 1.000 s] -

Questa istruzione tiene traccia di quanto tempo il suo contributo è stato vero. Se il suo ingresso è stato vero per almeno 1,000 s, allora l'uscita è vero. In caso contrario, l'uscita è falsa. L'ingresso non deve essere vero per 1000 millisecondi consecutivi, se l'ingresso diventa vero per 0,6 s, quindi false per 2,0 s, e quindi vale per 0,4 s, quindi l' uscita diventano vere. Dopo l'uscita diventa vera che rimarrà fedele anche dopo l'ingresso diventa falsa, finché l'ingresso è stato vero per più di 1,000 s. Questo timer deve quindi essere ripristinato manualmente, con l'operazione Resetta.

I `**tNome** 'conta variabile fino da zero in unità di tempi di scansione. Il Istruzione TON restituisce true quando la variabile contatore è maggiore o uguale al ritardo determinato. È possibile manipolare la variabile contatore altrove, per esempio con un'istruzione MOV.

> **RESET Citems TRTO**

---- {} RES RES -------- {} ----

Questa operazione resetta un temporizzatore o di un contatore. Temporizzatori TON e TOF sono ripristina automaticamente quando il loro ingresso è falso o vero, così RES è non è richiesto per questi timer. Timer RTO e CTU / CTD contatori sono non si ripristina automaticamente, in modo che deve essere resettato a mano con un RES istruzioni. Quando l'ingresso è vera, il contatore o timer viene azzerato; quando l'ingresso è falsa, non si interviene. Questa istruzione deve essere l'istruzione più a destra nel suo ramo.

> **ONE-SHOT RISING _**

- [OSR_ /] -

Questa istruzione genera normalmente falso. Se l'istruzione di ingresso del è vero durante questa scansione ed era falso durante la scansione precedente allora l'uscita è vera. Si genera quindi un impulso di una scansione larga ogni fronte di salita del segnale di ingresso. Questa istruzione è utile se si desidera attivare eventi fuori il fronte di salita di un segnale.

> **ONE-SHOT CADUTA _**

- [OSF \ _] -

Questa istruzione genera normalmente falso. Se l'istruzione di ingresso del è falso durante questa scansione ed era vero durante la scansione precedente allora l'uscita è vera. Si genera quindi un impulso di una scansione larghezza su ciascun fronte di discesa del segnale di ingresso. Questa istruzione è utile se si desidera attivare gli eventi fuori dal fronte di discesa di un segnale.

> CORTO CIRCUITO, circuito aperto

---- + ---- + -------- + + ----

La condizione di uscita di un corto-circuito è sempre pari al suo condizione di ingresso. La condizione di uscita di un circuito aperto è sempre false. Questi sono per lo più utili per il debug.

> MASTER CONTROL RELAY

- {MASTER RLY} -

Per impostazione predefinita, il ramo di ingresso in condizione di ogni ramo è vero. Se un maestro relè di istruzione di controllo viene eseguita con un ramo di ingresso in condizione di false, il ramo di ingresso in condizioni di tutti i rami seguenti diventa false. Questo continuerà fino a quando il relè di controllo principale successiva istruzione è raggiunta (a prescindere dal ramo di ingresso in condizioni di tale

istruzione). Queste istruzioni devono quindi essere utilizzati in coppia:

uno per (forse condizionale) iniziare la sezione eventualmente-disabilitata, e uno per porvi fine.

> SPOSTA {destvar: = {} Tret: =}

- {123 MOV} - {} srcvar MOV -

Quando l'ingresso di questa istruzione è vera, imposta la data destinazione variabile pari alla variabile data sorgente o costante. Quando l'ingresso di questa istruzione è nulla di falso accade. È possibile assegnare ad ogni variabile con l'istruzione di movimento; questo include timer e variabili di stato del contatore, che può essere distingue per il principale `T 'o' C '. Ad esempio, un'istruzione

0 in movimento `Tretentive 'è equivalente a un'istruzione di reset (RES) per tale timer. Questa istruzione deve essere l'istruzione più a destra nel suo ramo.

> **Operazione aritmetica {ADD kay: = {} SUB CCNT: =}**

- {'A' + 10} - {CCNT - 10} -

> {MUL dest: =} {DIV dv: =}

- {Var * -990} - {dv / -10000} -

Quando l'ingresso di questa istruzione è vera, imposta la data destinazione variabile uguale all'espressione data. Gli operandi possono essere sia variabili (compresi timer e variabili contatore) o costanti. Queste istruzioni utilizzano la matematica a 16 bit con segno. Ricordare che il risultato viene valutato ogni ciclo quando la condizione di ingresso vero. Se si sta incrementare o decrementare una variabile (ad esempio, se

la variabile di destinazione è anche uno degli operandi), allora si probabilmente non si vuole che, in genere si usa un one-shot in modo che essa viene valutata solo sul fronte di salita o di discesa dell'ingresso

condizione. Tronca Divide, 8/3 = 2. Questa istruzione deve essere l'istruzione più a destra nel suo ramo.

> CONFRONTA [var ==] [var>] [1> =]

- [Var2] - [1] - [Ton] -

> [Var / =] [-4 <] [1 <=]

- [Var2] - [vartwo] - [Serie] -

Se l'ingresso di questa istruzione è falsa allora l'uscita è falsa. Se l'ingresso è vera allora l'uscita è vera se e solo se il dato condizione è vera. Questa istruzione può essere utilizzata per confrontare (uguale, è maggiore, è maggiore o uguale a, non è uguale, è minore, è inferiore o uguale a) una variabile a una variabile, o per confrontare una variabile a 16 bit con segno costante.

> CONTATORE Cname Cname

- [CTU> = 5] ---- [CTD> = 5] -

Un contatore viene incrementato (CTU, conteggio in avanti) o diminuisce (CTD, conteggio giù) il conteggio associato ad ogni fronte di salita dell'ingresso ramo condizione (cioè quello che la condizione di ingresso del ramo passa da falsa a true). La condizione di uscita dal contatore è vero se il contatore variabile è maggiore o uguale a 5, e false altrimenti. Il condizione di uscita suonato può essere vero anche se la condizione di ingresso è falso, dipende solo dalla variabile contatore. Si può avere il CTU e istruzioni CTD con lo stesso nome, al fine di incrementare e decrementare il contatore stesso. L'istruzione RES può azzerare un contatore, oppure è possibile eseguire le operazioni generali di variabili sul variabile di conteggio.

> CONTATORE CIRCOLARE Cname

- {} CTC 00:07 -

A lavora circolare banco come un contatore normale CTU, esclusi quelli dopo aver raggiunto il limite superiore, reimposta la variabile contatore a 0. Per esempio, il contatore mostrato sopra conterebbe 0, 1,

2, 4, 5, 6, 7, 0, 1, 2, 3, 4, 5, 6, 7, 0, 2, ... Ciò è utile in combinazione con istruzioni condizionali sulla variabile `Cname '; è possibile utilizzare questo come un sequencer. CTC orologio contatori sul fronte di salita bordo della condizione condizione del ramo di ingresso. Questa istruzione deve essere è l'istruzione più a destra nel suo ramo.

> SHIFT REGISTER {SHIFT REG}

- {3} reg0 .. -

Un registro a scorrimento è associato a un insieme di variabili. Per esempio, questo registro a scorrimento è associato con le variabili reg0 `',` reg1', Reg2 `'e` REG3'. L'ingresso al registro a scorrimento è `reg0 '. Su

ogni fronte di salita del ramo di ingresso in condizioni, il registro a scorrimento sarà scorrimento a destra. Ciò significa che viene assegnato `REG3: = reg2 ',` reg2: = reg1 '. e `reg1: = reg0 '. `Reg0 'rimane invariato. Uno spostamento di grandi dimensioni registro possono facilmente occupare molta memoria. Questa istruzione deve essere è l'istruzione più a destra nel suo ramo.

> Look-up table {dest: =}

- {LUT [i]} -

Un look-up table è un insieme ordinato di n valori. Quando il ramo-in condizione è vera, la variabile intera `dest 'è posta uguale al voce nella tabella di ricerca corrispondente alla variabile intera `I '. L'indice parte da zero, in modo da `i 'deve essere compreso tra 0 e (N-1). Il comportamento di questa istruzione non è definita se il indice è al di fuori di questo intervallo. Questa istruzione deve essere la più a destra istruzione nel suo ramo.

> SPAZIO PER TABELLA lineare a tratti {variabile-y: =}

- {PWL [variabile-x]} -

Questo è un buon modo per approssimare una funzione complessa o curva. Si potrebbe, per esempio, essere utile se si sta cercando di applicare una curva di calibrazione per convertire una tensione di uscita grezzo da un sensore in più unità di misura adeguata.

Si supponga che si sta tentando di approssimare una funzione che converte una variabile di ingresso intero, x, ad una variabile di uscita integer, y. Voi conoscere la funzione in più punti, ad esempio, si potrebbe sapere che

f (0) = 2

f (5) = 10

f (10) = 50

f (100) = 100

Ciò significa che i punti

(X0, y0) = (0, 2)

(X1, y1) = (5, 10)

(X2, y2) = (10, 50)

(X3, y3) = (100, 100)

si trovano su quella curva. È possibile inserire i 4 punti in una tabella associato all'istruzione lineare a tratti. La lineare a tratti istruzione esaminerà il valore della variabile-x, e impostare il valore di variabile-y. Essa fissa Yvar in modo tale che la curva lineare a tratti passerà attraverso tutti i punti che si danno, per esempio, se si imposta variabile-x = 10, quindi l'istruzione imposta variabile-y = 50.

Se si dà l'istruzione un valore di variabile-x che si trova tra due dei valori di x per cui si è dato che punti, poi il istruzioni imposterà variabile-y in modo che (variabile-x, variabile-y) si trova sul rettilineo linea che collega i due punti nella tabella. Per esempio, variabile-x = 55 fornisce un'uscita di variabile-y = 75. (I due punti nella tabella sono (10, 50) e (100, 100). 55 è a metà strada tra 10 e 100, e 75 è a metà strada tra 50 e 100, in modo (55, 75) si trova sulla linea che collega questi due punti.)

I punti devono essere specificati in ordine crescente dalla coordinata x. Essa potrebbe non essere possibile eseguire operazioni matematiche richieste per alcune tabelle di consultazione con 16 bit matematica intero, se questo è il caso, allora LDmicro vi avvertirà. Ad esempio, questa tabella di ricerca genererà un errore :

(X0, y0) = (0, 0)

(X1, y1) = (300, 300)

È possibile correggere questi errori, rendendo la distanza tra i punti in il tavolo più piccolo. Ad esempio, questa tabella è equivalente a quello di cui sopra, e non produce un errore:

(X0, y0) = (0, 0)

(X1, y1) = (150, 150)

(X2, y2) = (300, 300)

Si dovrebbe quasi mai essere necessario utilizzare più di cinque o sei punti. L'aggiunta di più punti rende il codice più grande e più lento a eseguire. Il comportamento se si passa un valore di variabile-x `'maggiore di maggiori coordinata x nella tabella o inferiore alla più piccola x coordinare nella tabella non è definito. Questa istruzione deve essere la istruzione più a destra nel suo ramo.

> A / D CONVERTER LEGGI Aname

- {Leggi ADC} -

LDmicro può generare il codice per utilizzare l'A / D converter integrato al microcontrollori alcuni. Se la condizione di ingresso a questa istruzione è vero, allora un singolo campione dal convertitore A / D viene acquisito e memorizzato nella variabile `Aname '. Questa variabile può essere successivamente manipolato con operazioni generali variabili (inferiore, superiore, aritmetica, e così via). Assegnazione di un PIN al `'variabile Axxx nel stesso modo in cui si dovrebbe assegnare un PIN a un ingresso o un'uscita digitale, facendo doppio clic su di esso nella lista nella parte inferiore dello schermo. Se la condizione di ingresso a questo ramo è falso allora la variabile `Aname ' rimane invariato.

Per tutti i dispositivi attualmente supportati, ingresso 0 volt corrisponde a una lettura ADC di 0, e un ingresso pari a Vdd (tensione di alimentazione) corrisponde ad una lettura di ADC 1023. Se si utilizza un AVR, poi collegare AREF a Vdd. È possibile utilizzare le operazioni aritmetiche per scalare la lettura a più unità di misura adeguata in seguito, ma ricordate che si utilizza la matematica intero. In generale, non tutti i perni saranno disponibili per l'uso con l'A / D converter. Il software non consente di assegnare non-A / D pin ha un ingresso analogico. Questa istruzione deve essere l'istruzione più a destra nel suo ramo.

> **SET PWM CICLO duty_cycle**

- {PWM 32,8 kHz} -

LDmicro in grado di generare codice per utilizzare la periferica PWM integrata in microcontrollori alcuni. Se la condizione di ingresso a questa istruzione è vero, allora il duty cycle del PWM periferica è impostato sul valore della variabile duty_cycle. Il ciclo di lavoro deve essere un numero tra 0 e 100; 0 corrisponde sempre a bassa, e 100 corrisponde sempre alta. (Se si ha familiarità con il modo in cui il PWM funziona periferici, poi notare che questo significa che LDmicro ridimensiona automaticamente il ciclo utile variabile da cento a periodi di clock PWM.)

È possibile specificare la destinazione frequenza PWM, in Hz.. La frequenza che specificati potrebbe non essere esattamente realizzabile, a seconda di come si divide in frequenza di clock del microcontrollore. LDmicro scegliere la frequenza più vicina ottenibile, se l'errore è grande allora vi avverto. Velocità maggiori

possono sacrificare risoluzione. Questa istruzione deve essere l'istruzione più a destra nel suo ramo. Il runtime logica ladder consuma un timer per misurare il ciclo tempo. Ciò significa che il PWM è disponibile solo su microcontrollori con almeno due timer adatti. PWM utilizza pin CCP2 (non CCP1) su chip PIC16 e OC2 (non OC1A) su AVR.

> MAKE saved_var PERSISTENTE

- {} PERSIST -

Quando il ramo di ingresso in condizioni di questa istruzione è vero, fa sì che il specificata variabile intera per essere automaticamente salvato nella memoria EEPROM. Che significa che il suo valore persistono, anche quando il micro perde potenza. Non vi è alcuna necessità di salvare esplicitamente la variabile di EEPROM; che avverrà automaticamente, ogni volta che cambia variabili. La variabile viene automaticamente caricati da EEPROM dopo il power-on reset. Se una variabile che cambia frequentemente viene reso persistente, allora l' EEPROM in micro può usurarsi rapidamente, perché solo

buono per un numero limitato (circa 100 000) numero di scritture. Quando il ramo-in la condizione è falsa, non accade nulla. Questa istruzione deve essere la istruzione più a destra nel suo ramo.

> UART (SERIALE) RICEZIONE var

- {RECV UART} -

LDmicro in grado di generare codice per utilizzare la UART integrata in alcuni microcontrollori. In AVR con UART più solo UART1 (non UART0) è supportato. Configurare la velocità di trasmissione utilizzando Impostazioni -> MCU Parametri. Alcune velocità di trasmissione potrebbe non essere realizzabile con alcune frequenze di cristallo; LDmicro avvisa se questo è il caso.

Se la condizione di ingresso per questa istruzione è falsa, allora niente accade. Se la condizione di ingresso è vera, allora questa istruzione cerca per ricevere un singolo carattere dalla UART. Se nessun carattere viene letto quindi la condizione di uscita è falsa. Se un carattere viene letto quindi il suo Valore ASCII è memorizzato in `var ', e la condizione di uscita è vera per un singolo ciclo PLC.

> UART (SERIAL) INVIA var

- {} UART SEND -

LDmicro in grado di generare codice per utilizzare le UART incorporate in taluni microcontrollori. Su AVRS con UART più solo UART1 (non UART0) è supportato. Configurare la velocità di trasmissione utilizzando Impostazioni -> MCU Parametri. Alcune velocità di trasmissione potrebbe non essere realizzabile con alcune frequenze di cristallo; LDmicro avvisa se questo è il caso.

Se la condizione di ingresso per questa istruzione è falsa, allora niente accade. Se la condizione di ingresso è vera, allora questa istruzione scrive un singolo carattere alla UART. Il valore ASCII del carattere di deve inviare in precedenza sono stati memorizzati in `var '. L'uscita condizione del ramo è vero se la UART è occupato (attualmente la trasmissione di un carattere), e false altrimenti.

Ricorda che i caratteri richiedere un certo tempo per la trasmissione. Controllare l'uscita condizione di questa istruzione affinché il primo carattere ha stato trasmesso prima di provare a inviare un secondo personaggio, o utilizzare un timer per inserire un ritardo tra caratteri. Devi solo portare l'ingresso vera

condizione (tenta di inviare un carattere) quando l'uscita la condizione è falsa (UART non è occupato). Indagare le istruzioni in formato stringa (successiva) prima di utilizzare questo istruzioni. L'istruzione stringa formattata è molto più facile da usare, ed è quasi certamente in grado di fare quello che vuoi.

> Stringa formattata SOPRA UART var

- {"Pressione: \ 3 \ r \ n"} -

LDmicro in grado di generare codice per utilizzare le UART incorporate in taluni microcontrollori. Su AVRS con UART più solo UART1 (non UART0) è supportato. Configurare la velocità di trasmissione utilizzando Impostazioni -> MCU Parametri. Alcune velocità di trasmissione potrebbe non essere realizzabile con alcune frequenze di cristallo; LDmicro avvisa se questo è il caso.

Quando il ramo-in condizione di questa istruzione passa da falsa a vero, inizia ad inviare una stringa intera attraverso la porta seriale. Se la stringa contiene la sequenza speciale `\ 3 ', poi quella sequenza verrà sostituito con il valore di `var ', che è automaticamente convertito in una stringa. La variabile verrà formattato a prendere esattamente 3 caratteri, ad esempio, se `var 'è uguale a 35, quindi la stringa esatta sarà stampato `Pressione: 35 \ r \ n '(notare l'extra spazio). Se invece `var 'sono stati pari a 1432, il comportamento sarebbe essere definito, perché 1432 ha più di tre cifre. Allora sarebbe necessario usare `\ 4 'posto.

Se la variabile potrebbe essere negativo, quindi usare `\-3d '(o` \-4d' ecc) invece. Questo farà sì che LDmicro per stampare uno spazio importante per numeri positivi, e un segno meno per i numeri negativi.

Se più istruzioni stringa formattata sono eccitati in una sola volta (O se viene eccitato prima un'altra completa), o se questi le istruzioni sono mescolati con le istruzioni TX UART, il comportamento non è definito.

È anche possibile utilizzare questa istruzione per stampare una stringa fissa, senza l'interpolazione valore di una variabile intera è nel testo che viene inviato su seriale. In tal caso semplicemente non comprendono la speciale Sequenza di.

Usare `\ \ 'per un backslash letterale. In aggiunta alla sequenza di escape per interpolare una variabile intera, il seguente controllo caratteri disponibili:

* \ R - ritorno a capo

* \ N - newline

* \ F - salto pagina

* \ B - backspace

* \ XAB - 0xAB carattere con valore ASCII (hex)

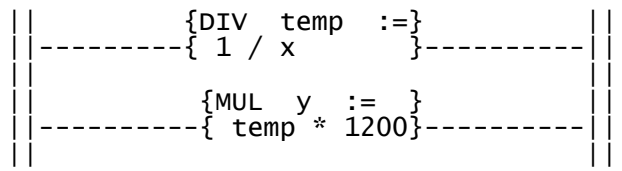
Il ramo di uscita condizione di questa istruzione è vera, mentre è la trasmissione dei dati, altrimenti falso. Questa istruzione consuma molto grande quantità di memoria di programma, quindi dovrebbe essere usato con parsimonia. Il implementazione attuale non è efficiente, ma una migliore volontà richiedere modifiche a tutti i back-end.

Una nota sull'uso di Math
==========

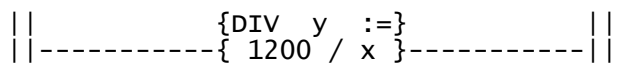
Ricorda che LDmicro esegue solo a 16 bit matematica intero. Ciò significa che il risultato finale di un calcolo che si deve effettuare un intero compreso tra -32768 e 32767. E 'anche significare che l'intermedio risultati del calcolo devono essere tutti in tale intervallo.

Per esempio, diciamo che si vuole calcolare y = (1 / x) * 1200, dove x è compreso tra 1 e 20. Allora y va tra il 1200 e il 60, che si inserisce in un intero a 16 bit, per cui è almeno in teoria possibile

eseguire il calcolo. Ci sono due modi in cui è possibile codificare questo: è possibile eseguire il reciproco, e quindi moltiplicare:



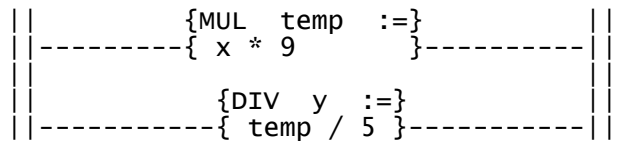
Oppure si può semplicemente fare la divisione diretta, in un unico passaggio:



Matematicamente, questi due sono equivalenti, ma se si tenta di loro, allora si troveranno che il primo dà un risultato errato di y = 0. Che perché underflow la variabile `temporanei '. Per esempio, quando x = 3,

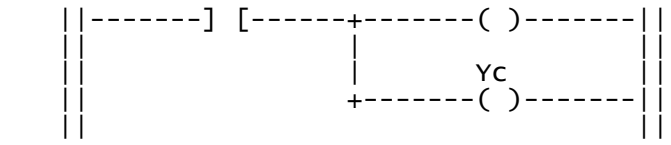
(1 / x) = 0,333, ma che non è un numero intero, l'operazione di divisione approssima questo come temp = 0. Allora y = temp * 1200 = 0. Nella seconda caso non c'è risultato intermedio di underflow, così tutto funziona. Se state vedendo problemi con la matematica, quindi controllare intermedio risultati (underflow o overflow, che `avvolge ', ad esempio, 32767 + 1 = -32768). Quando possibile, scegliere le unità che mettono i valori in una gamma da -100 a 100.

Quando avete bisogno di scalare una variabile da alcuni fattori, non utilizzando un multistrato e una divisione. Per esempio, in scala y = 1,8 * x, calcolare y = (9/5) * x (che è lo stesso, poiché 1,8 = 9/5), e questo codice come y = (9 * x) / 5, eseguendo prima moltiplicazione:

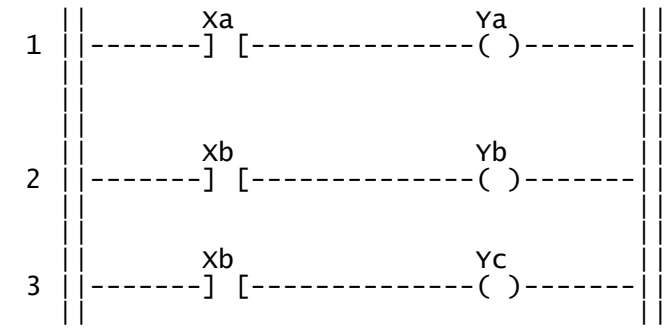


Questo funziona per tutte le x <(32767/9), o x <3640. Per i più grandi valori di x, la variabile `temp 'traboccherebbe. Vi è un limite inferiore simile a x.

STILE DI CODIFICA
==========
Consente più bobine in parallelo in un unico ramo. Ciò significa che si possono fare cose come questa:

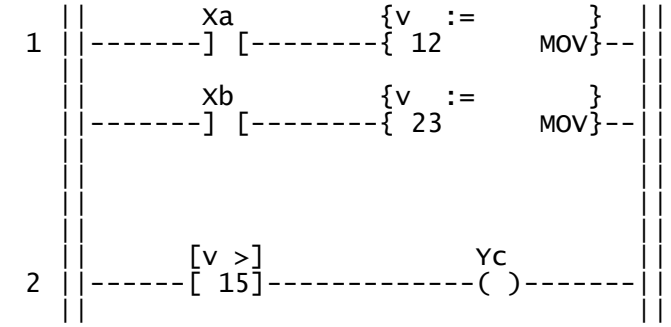


Invece che questa:

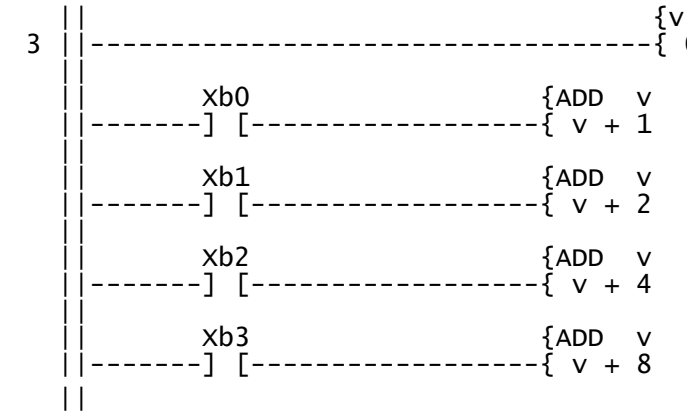


Ciò significa che, in teoria, si potrebbe scrivere un programma come un gradino gigante, e non vi è alcuna necessità di utilizzare più rami affatto. In pratica tale sarebbe una cattiva idea, perché, come rami diventano più complessi diventano più difficile da modificare senza cancellare e ridisegnare un sacco di logica. Ancora, è spesso una buona idea logica relativo gruppo insieme come una singola suonato. Questo genera il codice quasi identico a se hai fatto i rami separati, ma dimostra che essi sono correlati quando si guarda a loro sulla scala diagramma. ***

In generale, si ritiene forma poveri scrivere codice in modo che la sua uscita dipende dall'ordine dei pioli. Ad esempio, questo codice non è molto buona se sia Xa e Xb potrebbe mai essere vero:



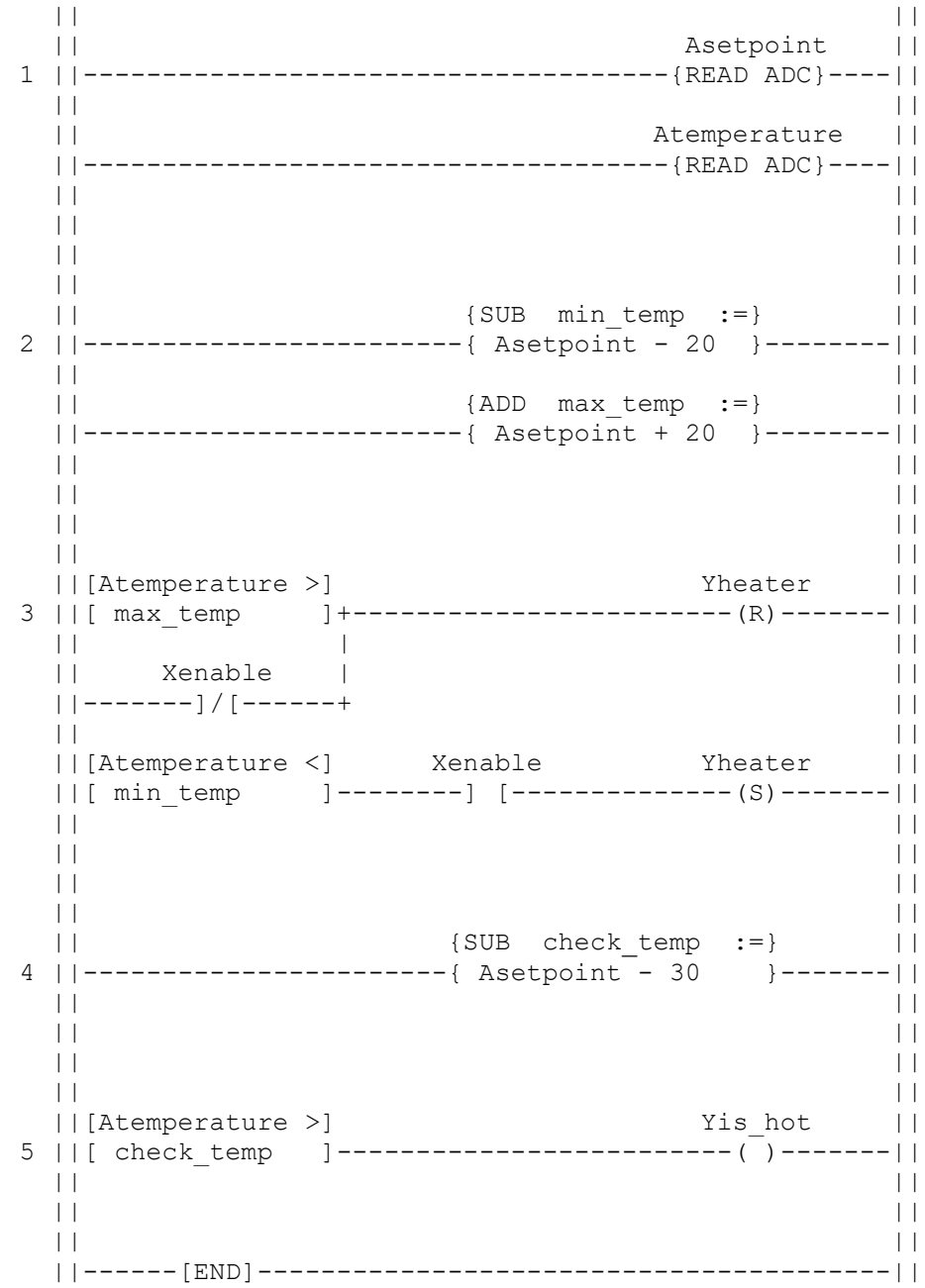
Spezzerò questa regola se così facendo posso fare un pezzo di codice molto più compatte, però. Ad esempio, ecco come vorrei convertire una quantità di 4-bit binario su xB3: 0 in un numero intero:



Jonathan Westhues

Esempio

Si tratta di un semplice pezzo di logica combinatoria. Ci sono tre termini di input, Xa, Xb e Xc. C'è un termine di uscita, Yout. L'espressione è Yout: = Xa e (Xb o (non Xc)). Questo ha senso se si pensa a Xa e Xb come contatti a relè normalmente aperti, Xc come contatti a relè normalmente chiusi, e yout come bobina di un relè. Naturalmente diventa più complicato di così:



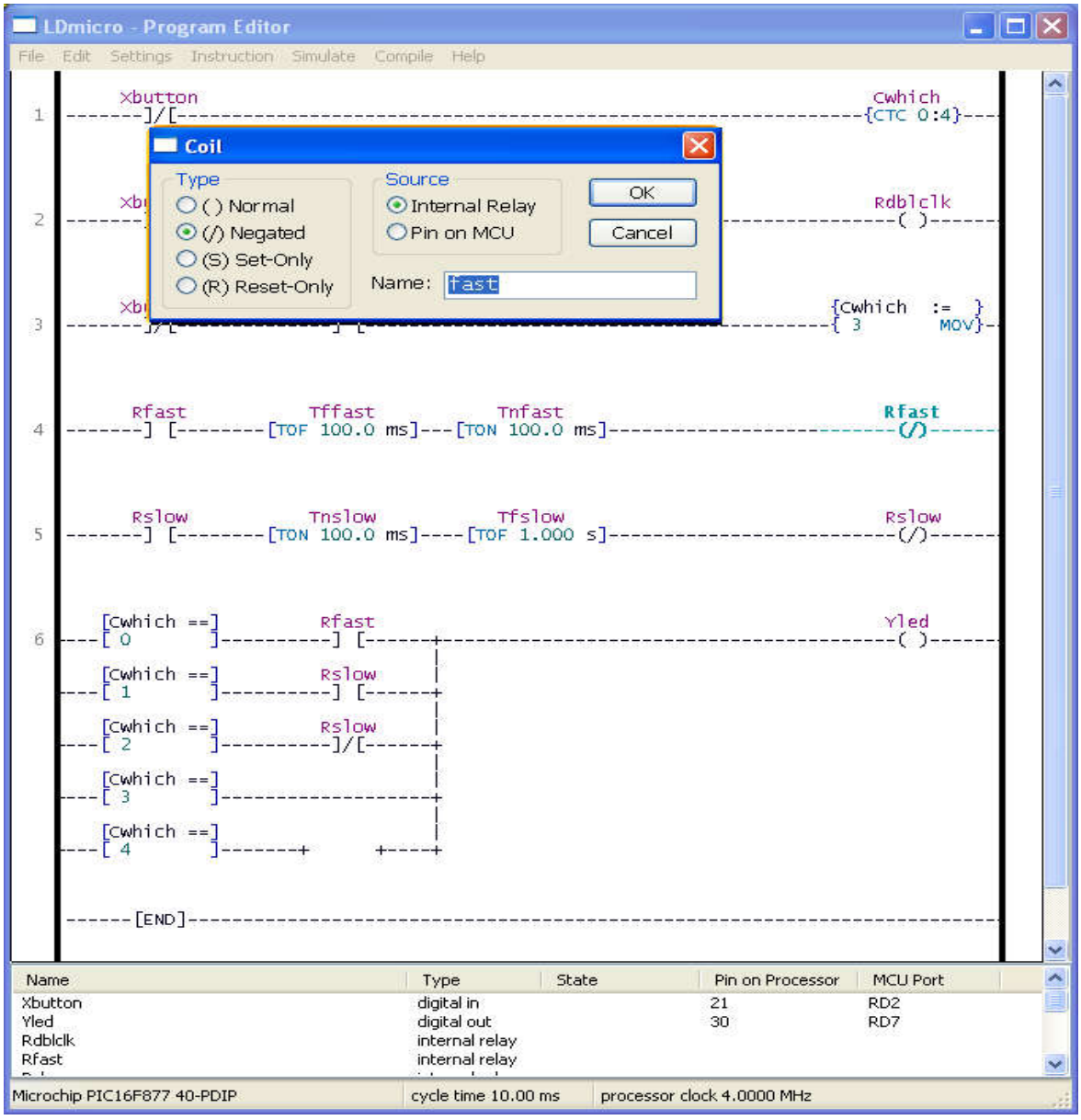
Questo è un semplice termostato. Ci sono due ingressi analogici, una delle quali è per il valore di riferimento, in modo che possa, ad esempio, essere collegato ad una pentola che, l'utilizzatore di selezionare la temperatura desiderata. L'altra fornisce la misura della temperatura, potrebbe essere un sensore di temperatura a semiconduttore, o un platino RTD adeguato circuito di interfaccia. C'è una uscita digitale, Yheater. Questo potrebbe controllare un elemento riscaldante, attraverso un interruttore adeguato (un

TRIAC, o un relè, o un relè a stato solido, o altro).

Chiudiamo il ciclo con un semplice isteretico (bang-bang) controller. Abbiamo selezionato più o meno 20 unità ADC di isteresi. Ciò significa che quando la temperatura scende al di sotto (setpoint - 20), si accende il riscaldamento, e quando sale sopra (valore nominale + 20), giriamo il riscaldatore.

Ho scelto di aggiungere una piccola pochi fronzoli. Primo, vi è un ingresso di abilitazione: la stufa viene forzato quando Xenable è basso. Ho anche aggiunto un indicatore luminoso, Yis_hot, per indicare che la temperatura è in regola. Ciò a fronte contro una soglia leggermente più fredda (setpoint - 20), in modo che la luce non lampeggia con il ciclo normale del termostato.

Questo è un esempio banale, ma deve essere chiaro che il linguaggio è molto espressivo. Logica ladder non è un linguaggio di programmazione, ma è Turing-completo, accettato nelle industrie, e, per un gruppo ristretto di (per lo più orientata al controllo) problemi, sorprendentemente conveniente.



LDmicro tutorial

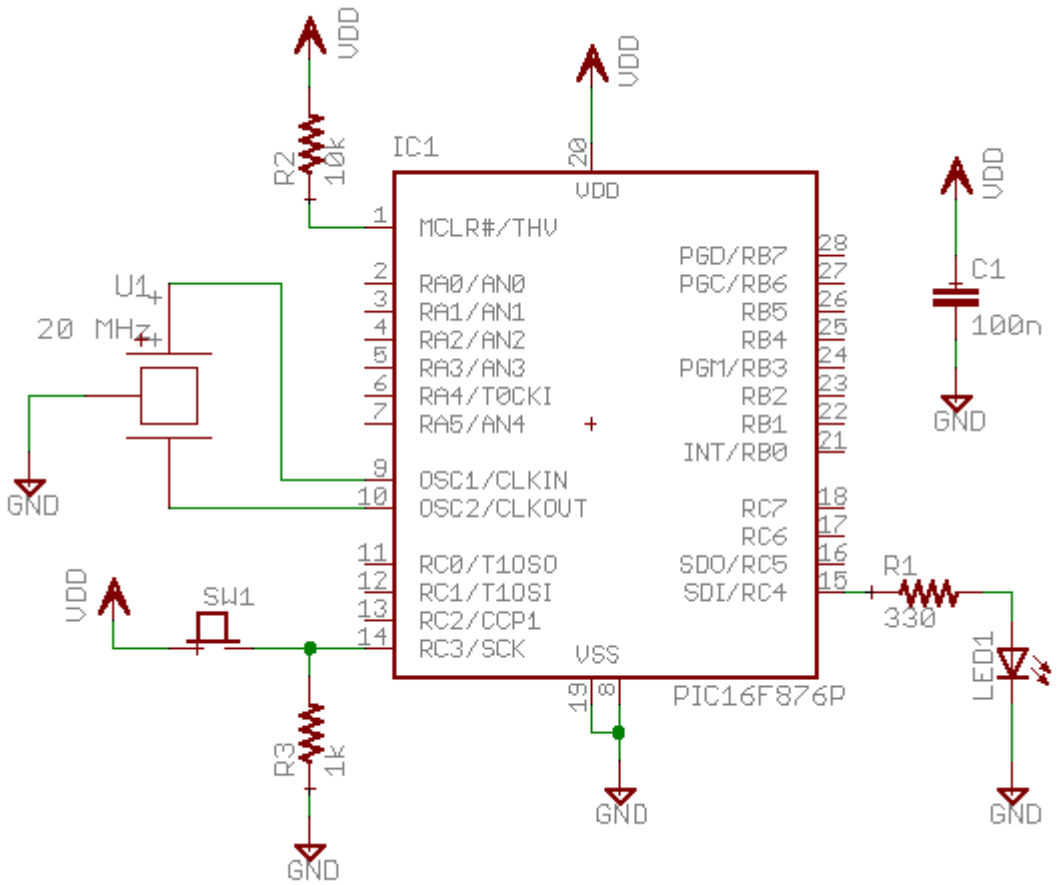
In questo tutorial, verrà mostrato come scrivere un semplice programma. Si assume che si abbia già scritto in ladder logic e che si abbiano alcune famigliarità di base con i micro-controllori, ma che non si abbia mai usato il programma LDmicro. Se non si conoscono molto bene gli argomenti riguardanti i ladder logic oppure PLCs, allora il plcs.net tutorial potrebbe essere d'aiuto.

Il nostro circuito avrà un pulsante ed un LED. All'accensione il LED sarà spento. Quando si presserà la prima volta il pulsante il LED si accenderà. Quando si premerà il pulsante una seconda volta il LED inizierà a lampeggiare. La terza volta che si preme il pulsante il LED si spegnerà di nuovo. Nella sequenza successiva, il ciclo verrà ripetuto.

Selezione del micro-controllore e schema

Verrà usato un PIC61F876, il quale è facilmente reperibile presso Digikey o altri distributori on-line. Il micro-controllore ha diverse forme; si è scelto il formato DIP.

Questo è il nostro schema:



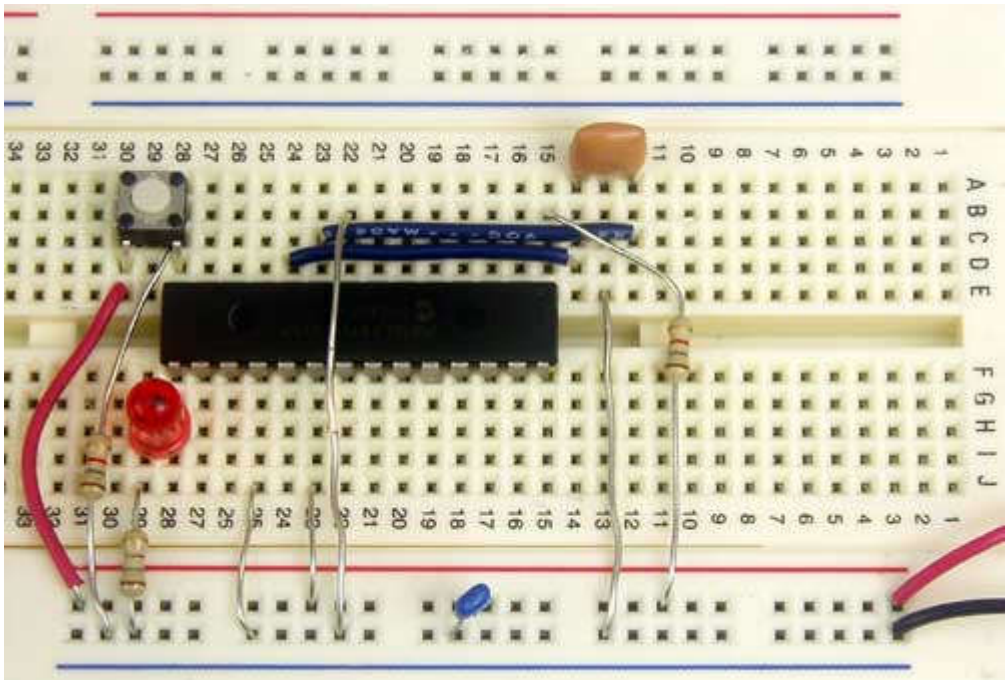
Il micro-controllore (IC1) il cui suo mumero identificativo è PIC16F876-20I/SP-ND presso Digikey. Quasi tutti i risuonatori a tre termilali (U1) andranno bene; si può provare

un 535-9356-ND oppure un X909-ND.

L'unica cosa che potrebbe confondere è che il pulsante *(pushbutton)* va alla Vdd, e che c'è un pull-down. Forse è più solito vedere un pulsante a ground con un pull-up. Per i TTL, questo ha importanza, invece, per i moderni CMOS non ne ha molta, e ho trovato, comunque, questo "attivo alto" *(active HIGH)* una disposizione meno confusionaria rispetto al tradizionale circuito basato in "attivo basso" *(active LOW)*

Si è anche scelto di adoperare un risuonatore ceramico provvisto di condensatori interni, U1, al posto di un quarzo e due condensatori ~20 pF. Un quarzo va senz'altro bene ed è più preciso, ma verrebbe ad costare un pò di più ed inoltre servirebbero altri componenti come, appunto, i due condensatori.

Il circuito si può costruire in molti differenti modi. Nel nostro caso si è preferito l'uso di una breadboard come quella che è, alla fine, a colleganemti effettuati, come quella mostrata nella foto:



(Il valore della resistenza mostrata non è abbastanza simile a quello dello schema; nessuno di loro è critico)

Ladder Diagram per il Programma

Primo, a noi servirà un oscillatore per generare il segnale lampeggiante per il LED. C'è un modo standard (schema fondamentale) per fare questo in ladder logic:

```
   ||      Rosc            Tosc_on         Tosc_off          Rosc       ||
 1 ||-------] [--------[TON 250.0 ms]---[TOF 250.0 ms]---------(/)-------||
```

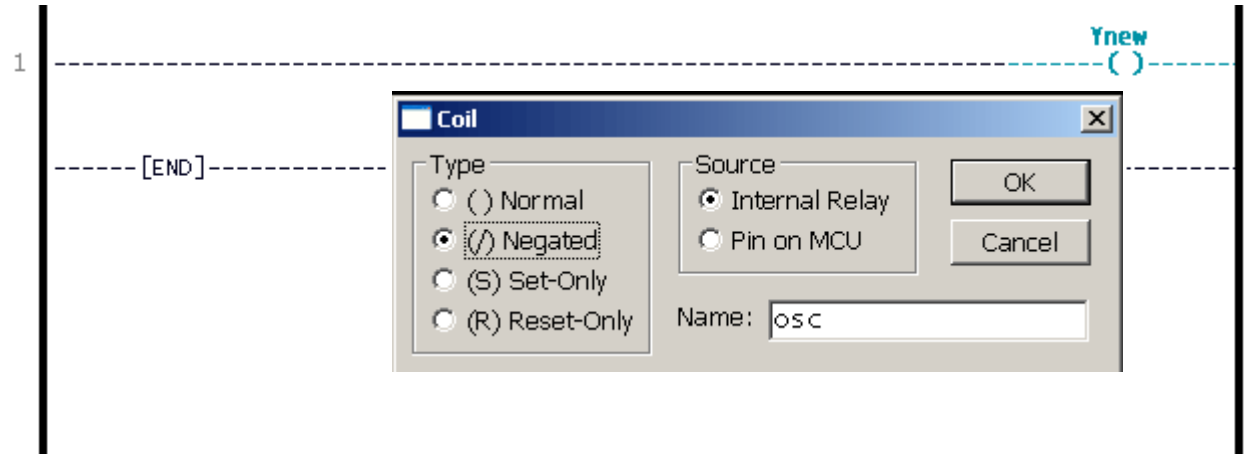
In breve, faremo in modo di usare le condizioni di stato del programma per far si che

queste siano quelle del LED, o meglio, degli stati del LED. E questo si può fare in questo modo:

```
   ||   [Cstate ==]                                          Yled       ||
 3 ||---[ 1       ]-------------------+-----------------------( )-------||
   ||                                  |                                 ||
   ||   [Cstate ==]         Rosc       |                                 ||
   ||---[ 2       ]----------] [------+                                 ||
```

E' abbastanza facile vedere attraverso il grafico che tutto andrà come volevamo. Se il programma è allo stato 1, allora l'istruzione 'Cstate'==1 attiverà 'Yled', come desideravamo. Nello stato 2, l'istruzione 'Cstate 2==' attiverà 'Yled', ma solo quando 'Rosc' è anche vero. Per cui tutte le volte che 'Rosc' oscilla, vorrà dire che il LED lampeggia, come desideravamo. Finalmente, nello stato 0, nessuna delle istruzioni di uguale è soddisfatta per la condizione vero, per cui non c'è nessun modo che 'Yled' si possa accendere.

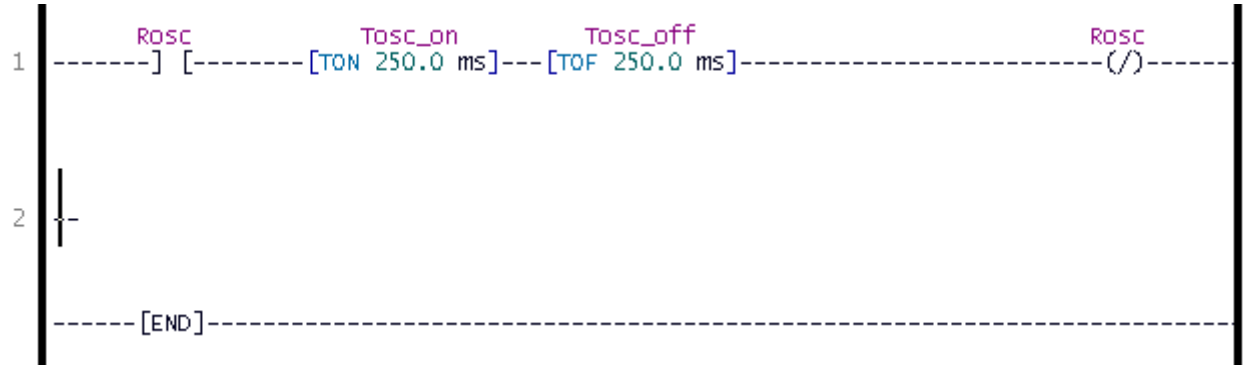
Vogliamo adesso inserire la prima rung in relazione a quanto detto più sopra. Inizieremo con la bobina 'coil', per cui selezioneremo Instruction -> Insert Coil. Con questa operazione avremo inserito nella riga del nostro foglio di lavoro, un simbolo convenzionale di una bobina a cui, di default, è dato il nome di 'Ynew'. Questo era ciò che si voleva eccetto il nome e che deve essere 'negated'. Quindi, per introdurre i valori che si desiderano, occorre dare doppio click sul simbolo per fare apparire una finestra di dialogo dove possiamo configurare appropriatamente la nostra bobina:



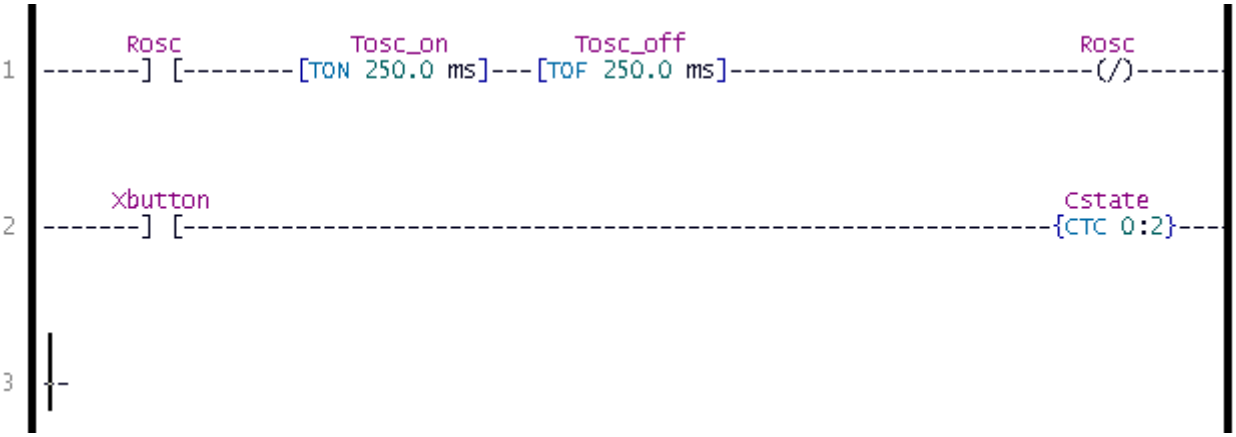
Adesso possiamo completare il resto del rung inserendo, allo stesso modo, gli altri componenti, nel nostro caso, i simboli. Clicca su il fronte sinistro della bobina in modo tale che il cursore si ponga, rispetto al simbolo, in verticale e (importante), alla sinistra del

simbolo. Adesso selezione Instruction -> Insert TON (Delayed Turn On). Ancora una volta doppio click sul timer per far apparire la finestra di dialogo e inseriremo il nome "on" ed un periodo di ritardo di 250 ms. Aggiungiamo allo stesso modo 'TOF' (nome 'OF' e delay 250 ms). In ultimo portandoci alla sinistra di 'TON' aggiungiamo dei contatti.

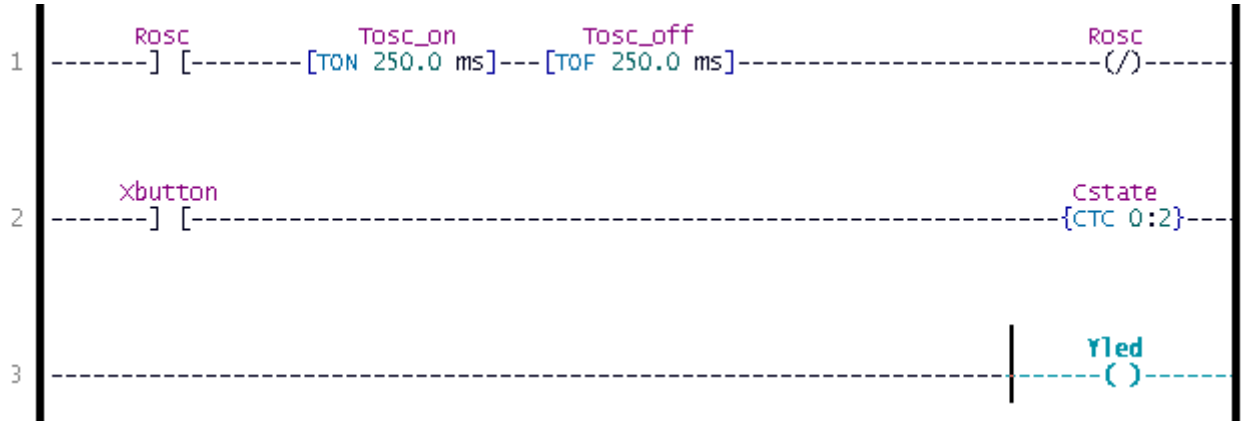
Adesso, per completare il nostro schema, vogliamo inserire un secondo rung, così, selezioniamo Edit -> Insert Rung After. Adesso portati con l'indicatore del mouse proprio sul secondo rung appena creato e clicca su, il cursore si porterà proprio lì:



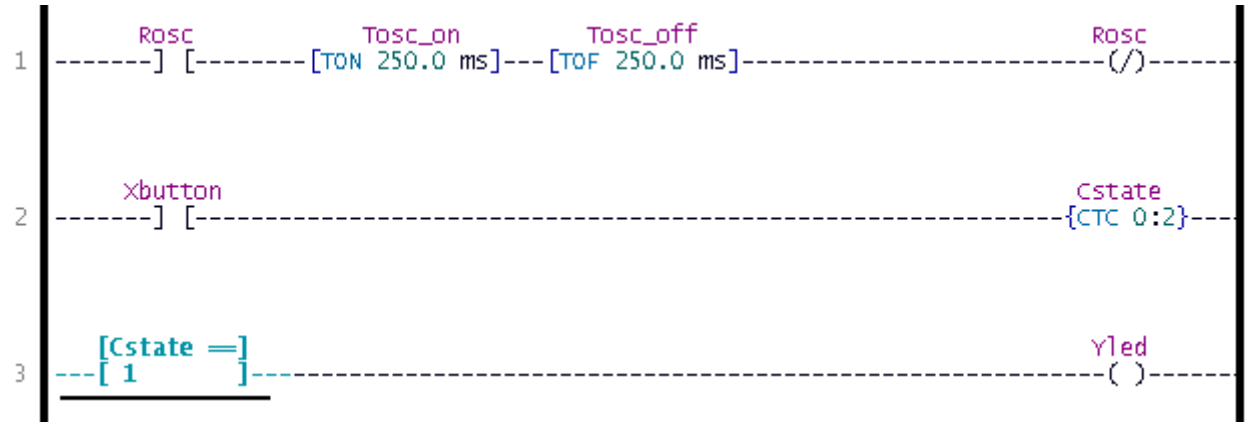
Il secondo rung è facile: basta completare due istruzioni nel giusto ordine mettendo il cursore dove si vuole e quindi selezionare Instruction -> Insert ....contact....ricordarsi di assegnare il nome ('Xbotton') ai contatti poi , allo stesso modo, selezionare Instruction -> Insert...CTC (Count Circular) ... assegnare un nome (ad esempio "state" e nel MAX VALUE inserire 2 (infatti 0, 1, 2, ...0, 1, 2, ricordate?) che è il limite massimo del contatore. A questo punto, di nuovo, seleziona Edit -> Insert Rung After. Il foglio di lavoro dovrebbe presentarsi così:



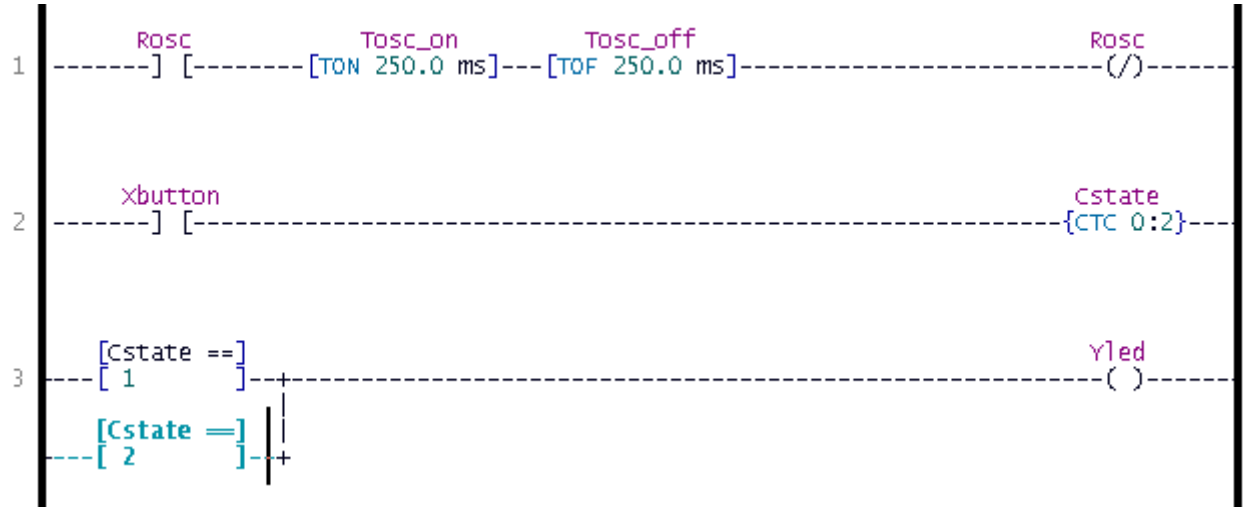
Il terzo rung sarà un pò più "furfante", perché conterrà delle ramificazioni in parallelo. Questo vuol dire che bisogna considerare l'ordine nel quale si inseriscono le istruzioni. Primo, inserire il segno grafico della bobina *(coil)* ed assegnare un nome, nel nostro esempio"name"= Led "tipo"=normale:



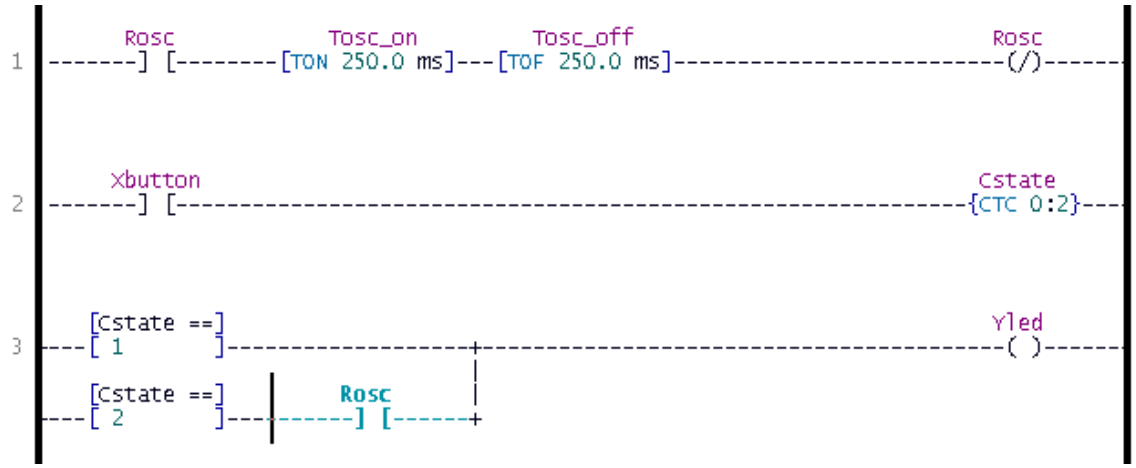
Adesso inseriamo la prima istruzione "Equal" alla sinistra della nostra coil che avevamo appena inserito, come al solito, e attraverso la maschera di dialogo inseriamo le giuste variabili del nome e del valore. Dopo che abbiamo fatto questo, precederemo ad aggiungere il ramo posto in parallelo: cliccando sulla parte bassa del simbolo "uguale" il cursore si disporrà in orizzontale al simbolo stesso dell'istruzione di "equal".



A questo punto, con il cursore posto lungo il lato basso del simbolo selezioneremo Instruction -> Insert EQU (Compare for Equals). Da notare che sino a quando il cursore è sotto la prima istruzione di "equal", l'istruzione successiva si verrà a disporre in parallelo ad essa. A questo punto, come di solito, rinomineremo ed aggiorneremo i valori della variabile. Per finire il rung dovremo inserire i contatti "Rosc" alla destra della seconda istruzione di "equal". Per fare questo, bisogna cliccare sul lato destro della seconda istruzione di "equal":

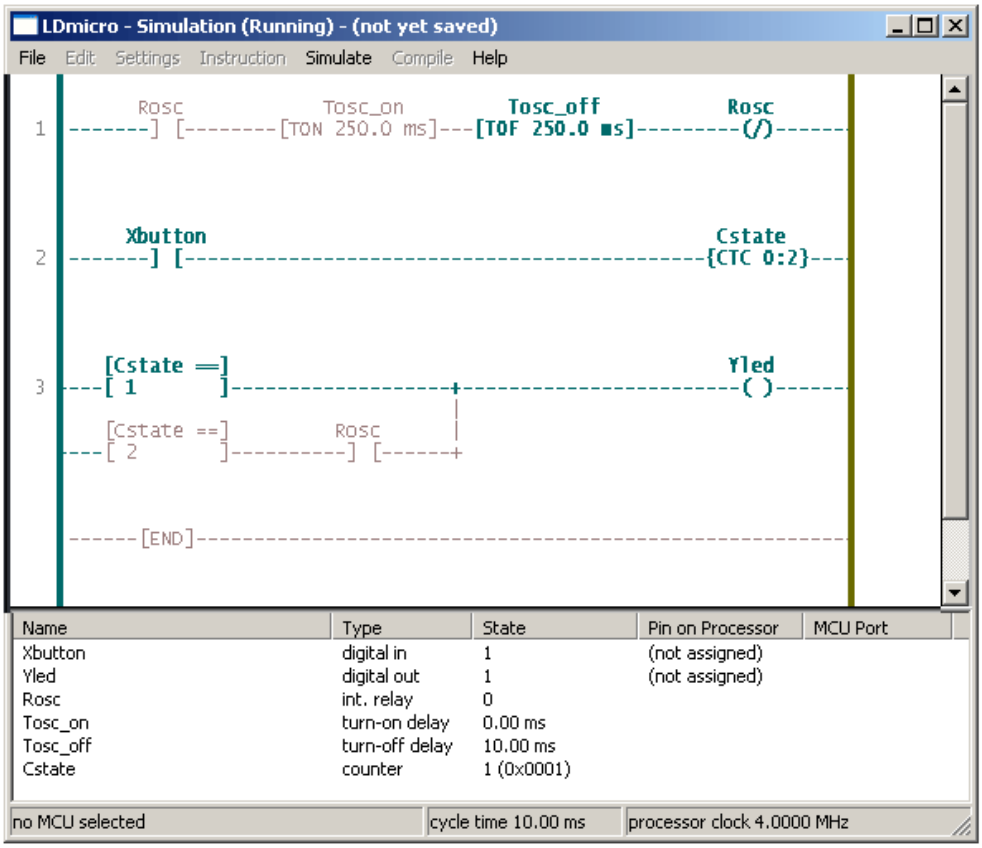


A questo punto selezionare Instruction -> Insert Coil; il simbolo di coil verrà inserito in serie alla seconda istruzione, come da noi richiesto. Rinomina ed attribuisci i giusti valori, e con questo hai concluso la fase di programmazione.



Simulando il Programma

Adesso siamo pronti per simulare il nostro circuito. Selezione Simulate -> Simulation Mode. Lo schermo cambierà; e il grafico ladder apparirà prevalentemente grigio, e non si vedranno, al momento, che le cose cambino anche perché il PLC non è ancora sottoposto ai cicli. Per lanciare i cicli bisogna selezionare Simulate -> Start Real-Time Simulation. Fatto questo, si può vedere che qualcosa cambia, l'oscillatore è in esecuzione ma il LED ('Yled') resta spento, come d'altra parte avevamo progettato, perché ancora non è stato pigiato il pulsante. Per simulare la pressione sul pulsante, perciò, fare doppio click sul testo 'Xbotton' nella lista che appare nella sotto finestra in fondo allo schermo. Con questa operazione si è simulato di portare il pulsante a livello alto *(input high)* ; Questo è quello che accadrà se si tiene mantenuto pressato il pulsante senza rilasciarlo. Si può vedere che il programma lavora: il contatore 'Cstate' è, adesso, uguale ad 1, il che corrisponde allo stato 'steady on', che è quello che volevamo.



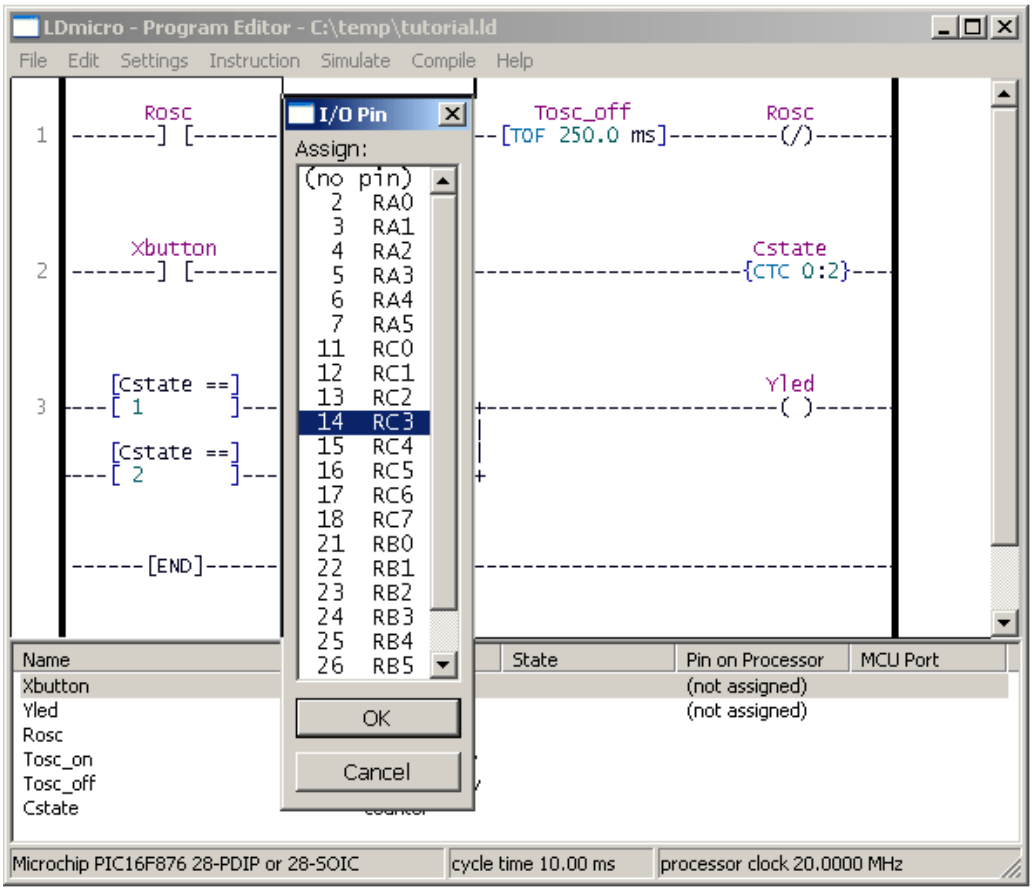
Il LED output è high; si può vedere che il suo valore è 1 nella lista, e il 'Yled' appare red nel grafico. Doppio clic sulla scritta 'Xbotton' nella solita list in fondo al monitor, e quindi doppio click ancora per simulare il rilascio del pulsante. La bobina 'Yled' inizierà a lampeggiare, come desiderato. Se si simula la terza pressione si vedrà che l'output andrà steady low *basso.*

Compilando l'IHEX File

Adesso siamo abbastanza sicuri che il programma lavora. A questo punto siamo pronti per generare il codice attuale, e provarlo nel micro. Innanzi tutto usciamo dal modo simulazione de-selezionando dal menu principale Simulate -> Simulation Mode, oppure Escape.

Subito dopo dobbiamo selezionare un micro-controllore. Prima avevamo deciso che avremmo usato un PIC16F876, per cui selezioniamo Settings -> Microcontroller -> Microchip PIC16F876 28-PDIP oppure 28-SOIC. Inoltre, dobbiamo ancora dire al pregramma LDmicro che tipo di quarzo useremo e quale sarà cycle time. Selezioniamo, allora, Settings -> MCU Parameters e completiamo il clock speed con of 20 MHz.. Lasciamo il cycle time a 10 ms; questo, usualmente, è un buon valore.

Adesso possiamo assegnare i pins ai nostri input ed output. Doppio click si 'Xbotton' nella lista in basso allo schermo, e scegliamo pin 14 del PIC, che corrisponde alla porta RC3 della MCU. (Non c'è nessuna ragione per prendersi cura di quale porta si userà; basta guardare ai numeri dei pin).



Clicca 'OK' e quindi ripeti il procedimanto per 'Yled' che, come si può vedere dallo schema elettrico, va al pin 15. Gli altri elementi contenuti nella lista sono variabili interne e bits in memoria, per cui non c'è nessun bisogno di assegnare a loro dei pins. LDmicro allocherà memoria per loro quando si andrà a compilare.

Adesso si è pronti per compilare. Seleziona Compile -> Compile, e specificare dove si vuole mettere l'IHEX file. Quindi, usare qualunque tipo di dispositivo di programmazione si ha a disposizione per trasferire dentro il micro il codice, ed adesso si è pronti per provare il nostro circuito.

Questo completa il mio tutorial. E' possibile scrivere programmi molto più complessi di questo, naturalmente. Un programma come questo usa veramente una frazione di poca memoria rispetto a quella disponibile nella memoria del micro, per cui c'è abbastanza spazio per molti altri rungs di logica. LDmicro offre anche istruzioni specializzate per cose come aritmetica, analogici input ed output, PWM, e anche caratteri da inviare a dei LCD. Consultare il manuale per ulteriori dettagli.

```
LDmicro0.1
MICRO=Microchip PIC16F876 28-PDIP or 28-SOIC
CYCLE=10000
CRYSTAL=20000000
BAUD=2400

IO LIST
    Xbutton at 14
    Yled at 15
END

PROGRAM
RUNG
    CONTACTS Rosc 0
    TON Tosc_on 250000
    TOF Tosc_off 250000
    COIL Rosc 1 0 0
END
RUNG
    CONTACTS Xbutton 0
    CTC Cstate 2
END
RUNG
    PARALLEL
        EQU Cstate 1
        SERIES
            EQU Cstate 2
            CONTACTS Rosc 0
        END
    END
    COIL Yled 0 0 0
END
```

Latch Instructions

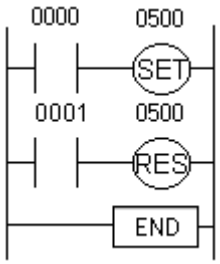
Now that we understand how inputs and outputs are processed by the plc, let's look at a variation of our regular outputs. Regular output coils are of course an essential part of our programs but we must remember that they are only TRUE when ALL INSTRUCTIONS before them on the rung are also TRUE. What happens if they are not? Then of course, the output will become false.(turn off)

Think back to the lunch bell example we did a few chapters ago. What would've happened if we couldn't find a "push on/push off" switch? Then we would've had to keep pressing the button for as long as we wanted the bell to sound. (A momentary switch) The latching instructions let us use momentary switches and program the plc so that when we push one the output turns on and when we push another the output turns off.

Maybe now you're saying to yourself "What the heck is he talking about?". (It's also what I'm thinking!) So let's do a real world example.

Picture the remote control for your TV. It has a button for ON and another for OFF. (mine does, anyway) When I push the ON button the TV turns on. When I push the OFF button the TV turns off. I don't have to keep pushing the ON button to keep the TV on. This would be the function of a latching instruction.

The latch instruction is often called a SET or OTL (output latch). The unlatch instruction is often called a RES (reset), OUT (output unlatch) or RST (reset). The diagram below shows how to use them in a program.



Here we are using 2 momentary push button switches. One is physically connected to input 0000 while the other is physically connected to input 0001. When the operator pushes switch 0000 the instruction "set 0500" will become true and output 0500 physically turns on. Even after the operator stops pushing the switch, the output (0500) will remain on. It is latched on. The only way to turn off output 0500 is turn on input 0001. This will cause the instruction "res 0500" to become true thereby unlatching or resetting output 0500.

Here's something to think about. What would happen if input 0000 and 0001 both turn on at the exact same time.

Will output 0500 be latched or unlatched?

To answer this question we have to think about the scanning sequence. The ladder is always scanned from top to bottom, left to right. The first thing in the scan is to physically look at the inputs. 0000 and 0001 are both physically on. Next the plc executes the program. Starting from the top left, input 0000 is true therefore it should set 0500. Next it goes to the next rung and since input 0001 is true it should reset 0500. The last thing it said was to reset 0500. Therefore on the last part of the scan when it updates the outputs it will keep 0500 off. (i.e. reset 0500)

Counters

A counter is a simple device intended to do one simple thing - count. Using them, however, can sometimes be a challenge because every manufacturer (for whatever reason) seems to use them a different way. Rest assured that the following information will let you simply and easily program anybody's counters.

What kinds of counters are there? Well, there are up-counters (they only count up 1,2,3...). These are called CTU,(count up) CNT,C, or CTR. There are down counters (they only count down 9,8,7,...). These are typically called CTD (count down) when they are a separate instruction. There are also up-down counters (they count up and/or down 1,2,3,4,3,2,3,4,5,...) These are typically called UDC(up-down counter) when they are separate instructions.

Many manufacturers have only one or two types of counters but they can be used to count up, down or both. *Confused yet?* Can you say "no standardization"? Don't worry, the theory is all the same regardless of what the manufacturers call them. A counter is a counter is a counter...

To further confuse the issue, most manufacturers also include a limited number of high-speed counters. These are commonly called HSC (high-speed counter), CTH (CounTer High-speed?) or whatever. Typically a high-speed counter is a "*hardware*" device. The normal counters listed above are typically "*software*" counters. In other words they don't physically exist in the plc but rather they are simulated in software. Hardware counters do exist in the plc and they are not dependent on scan time. **A good rule of thumb** is simply to always use the normal (software) counters unless the pulses you are counting will arrive faster than 2X the scan time. (i.e. if the scan time is 2ms and pulses will be arriving for counting every 4ms or longer then use a software counter. If they arrive faster than every 4ms (3ms for example) then use the hardware (high-speed) counters. (2xscan time = 2x2ms= 4ms)

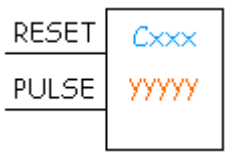
To use them we must know 3 things:

1. Where the pulses that we want to count are coming from. Typically this is from one of the inputs.(a sensor connected to input 0000 for example)
2. How many pulses we want to count before we react. Let's count 5 widgets before we box them, for example.
3. When/how we will reset the counter so it can count again. After we count 5 widgets lets reset the counter, for example.

When the program is running on the plc the program typically displays the current or "*accumulated*" value for us so we can see the current count value.

Typically counters can count from 0 to 9999, -32,768 to +32,767 or 0 to 65535. Why the weird numbers? Because most PLCs have 16-bit counters. We'll get into what this means in a later chapter but for now suffice it to say that 0-9999 is 16-bit BCD (binary coded decimal) and that -32,768 to 32767 and 0 to 65535 is 16-bit binary.

Here are some of the instruction symbols we will encounter (depending on which manufacturer we choose) and how to use them. Remember that while they may look different they are all used basically the same way. If we can setup one we can setup any of them.



In this counter we need 2 inputs.

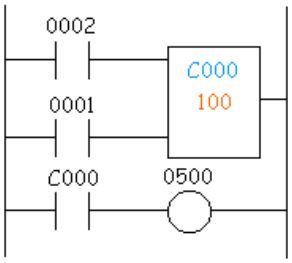
One goes before the reset line. When this input turns on the current (accumulated) count value will return to 0. The second input is the address where the pulses we are counting are coming from.

For example, if we are counting how many widgets pass in front of the sensor that is physically connected to input 0001 then we would put normally open contacts with the address 0001 in front of the pulse line.

Cxxx is the name of the counter. If we want to call it counter 000 then we would put "C000" here.

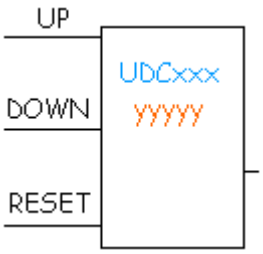
yyyyy is the number of pulses we want to count before doing something. If we want to count 5 widgets before turning on a physical output to box them we would put 5 here. If we wanted to count 100 widgets then we would put 100 here, etc. When the counter is finished (i.e we counted yyyyy widgets) it will turn on a separate set of contacts that we also label Cxxx.

Note that the counter accumulated value ONLY changes at the off to on transition of the pulse input.

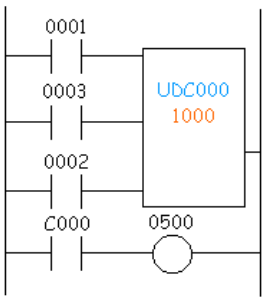


Here's the symbol on a ladder showing how we set up a counter (we'll name it counter 000) to count 100 widgets from input 0001 before turning on output 500. Sensor 0002 resets the counter.

Below is one symbol we may encounter for an up-down counter. We'll use the same abbreviation as we did for the example above.(i.e. UDCxxx and yyyyy)



In this up-down counter we need to assign 3 inputs. The reset input has the same function as above. However, instead of having only one input for the pulse counting we now have 2. One is for counting up and the other is for counting down. In this example we will call the counter UDC000 and we will give it a preset value of 1000. (we'll count 1000 total pulses) For inputs we'll use a sensor which will turn on input 0001 when it sees a target and another sensor at input 0003 will also turn on when it sees a target. When input 0001 turns on we count up and when input 0003 turns on we count down. When we reach 1000 pulses we will turn on output 500. Again note that the counter accumulated value ONLY changes at the off to on transition of the pulse input. The ladder diagram is shown below.



Timers

Let's now see how a timer works. What is a timer? Its exactly what the word says... it is an instruction that waits a set amount of time before doing something. Sounds simple doesn't it.

When we look at the different kinds of timers available the fun begins. As always, different types of timers are available with different manufacturers. Here are most of them:

- **On-Delay timer**-This type of timer simply "delays turning on". In other words, after our sensor (input) turns on we wait x-seconds before activating a solenoid valve (output). This is the most common timer. It is often called TON (timer on-delay), TIM (timer) or TMR (timer).
- **Off-Delay timer**- This type of timer is the opposite of the on-delay timer listed above. This timer simply "delays turning off". After our sensor (input) sees a target we turn on a solenoid (output). When the sensor no longer sees the target we hold the solenoid on for x-seconds before turning it off. It is called a TOF (timer off-delay) and is less common than the on-delay type listed above. (i.e. few manufacturers include this type of timer)
- **Retentive or Accumulating timer**- This type of timer needs 2 inputs. One input starts the timing event (i.e. the clock starts ticking) and the other resets it. The on/off delay timers above would be reset if the input sensor wasn't on/off for the complete timer duration. This timer however holds or retains the current elapsed time when the sensor turns off in mid-stream. For example, we want to know how long a sensor is on for during a 1 hour period. If we use one of the above timers they will keep resetting when the sensor turns off/on. This timer however, will give us a total or accumulated time. It is often called an RTO (retentive timer) or TMRA (accumulating timer).

Let's now see how to use them. We typically need to know 2 things:

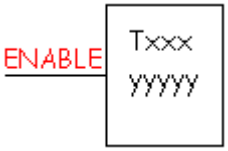
1. **What will enable the timer**. Typically this is one of the inputs.(a sensor connected to input 0000 for example)
2. **How long we want to delay before we react**. Let's wait 5 seconds before we turn on a solenoid, for example.

When the instructions before the timer symbol are true the timer starts "ticking". When the time elapses the timer will automatically close its contacts. When the program is running on the plc the program typically displays the elapsed or "*accumulated*" time for us so we can see the current value. Typically timers can tick from 0 to 9999 or 0 to 65535 times.

Why the weird numbers? Again its because most PLCs have 16-bit timers. We'll get into what this means in a later chapter but for now suffice it to say that 0-9999 is 16-bit BCD (binary coded decimal) and that 0 to 65535 is 16-bit binary. Each tick of the clock is equal to x-seconds.

Typically each manufacturer offers several different ticks. Most manufacturers offer 10 and 100 ms increments (ticks of the clock). An "**ms**" is a milli-second or *1/1000th* of a second. Several manufacturers also offer 1ms as well as 1 second increments. These different increment timers work the same as above but sometimes they have different names to show their timebase. Some are TMH (high speed timer), TMS (super high speed timer) or TMRAF (accumulating fast timer)

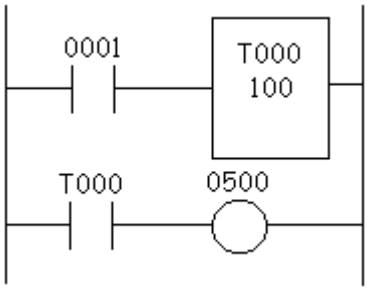
Shown below is a typical timer instruction symbol we will encounter (depending on which manufacturer we choose) and how to use it. Remember that while they may look different they are all used basically the same way. If we can setup one we can setup any of them.



This timer is the on-delay type and is named Txxx. When the enable input is on the timer starts to tick. When it

ticks yyyyy (the preset value) times, it will turn on its contacts that we will use later in the program. Remember that the duration of a tick (increment) varies with the vendor and the timebase used. (i.e. a tick might be 1ms or 1 second or...)

Below is the symbol shown on a ladder diagram:

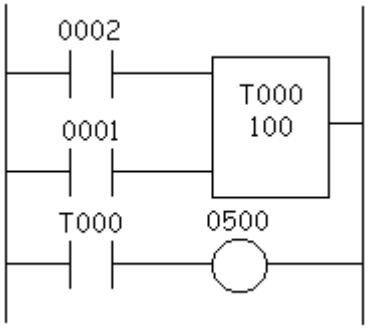


In this diagram we wait for input 0001 to turn on. When it does, timer T000 (a 100ms increment timer) starts ticking. It will tick 100 times. Each tick (increment) is 100ms so the timer will be a 10000ms (i.e. 10 second) timer. 100ticks X 100ms = 10,000ms. When 10 seconds have elapsed, the T000 contacts close and 500 turns on. When input 0001 turns off(false) the timer T000 will reset back to 0 causing its contacts to turn off(become false) thereby making output 500 turn back off.

An accumulating timer would look similar to this:

This timer is named Txxx. When the enable input is on the timer starts to tick. When it ticks yyyyy (the preset value) times, it will turn on its contacts that we will use later in the program. Remember that the duration of a tick (increment) varies with the vendor and the timebase used. (i.e. a tick might be 1ms or 1 second or...) If however, the enable input turns off before the timer has completed, the current value will be retained. When the input turns back on, the timer will continue from where it left off. The only way to force the timer back to its preset value to start again is to turn on the reset input.

The symbol is shown in the ladder diagram below.

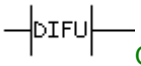


In this diagram we wait for input 0002 to turn on. When it does timer T000 (a 10ms increment timer) starts ticking. It will tick 100 times. Each tick (increment) is 10ms so the timer will be a 1000ms (i.e. 1 second) timer. 100ticks X 10ms = 1,000ms. When 1 second has elapsed, the T000 contacts close and 500 turns on. If input 0002 turns back off the current elapsed time will be retained. When 0002 turns back on the timer will continue where it left off. When input 0001 turns on (true) the timer T000 will reset back to 0 causing its contacts to turn off (become false) thereby making output 500 turn back off.

One-shots

A one-shot is an interesting and invaluable programming tool. At first glance it might be difficult to figure out why such an instruction is needed. After we understand what this instruction does and how to use it, however, the necessity will become clear.

A one-shot is used to make something happen for **ONLY 1 SCAN**. (you do remember what a scan is, right??) Most manufacturers have one-shots that react to an off to on transition and a different type that reacts to an on to off transition. Some names for the instructions could be difu/difd (differentiate up/down), sotu/sotd (single output up/down), osr (one-shot rising) and others. They all, however, end up with the same result regardless of the name.

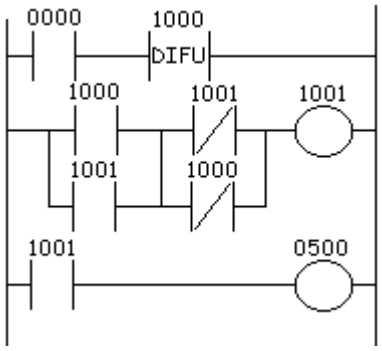


One-shot Instruction

Above is the symbol for a difu (one-shot) instruction. A difd looks the same but inside the symbol it says "difd". Some of the manufacturers have it in the shape of a box but, regardless of the symbol, they all function the same way. For those manufacturers that don't include a differentiate down instruction, you can get the same effect by putting a NC (normally closed) instruction before it instead of a NO(normally open) instruction. (i.e. reverse the logic before the difu instruction)

Let's now setup an application to see how this instruction actually functions in a ladder. This instruction is most often used with some of the advanced instructions where we do some things that MUST happen only once. However, since we haven't gotten that far yet, let's set up a flip/flop circuit. In simple terms, a flip/flop turns something around each time an action happens. Here we'll use a single pushbutton switch. The first time the operator pushes it we want an output to turn on. It will remain "latched" on until the next time the operator pushes the button. When he does, the output turns off.

Here's the ladder diagram that does just that:



Now this looks confusing! Actually it's not if we take it one step at a time.

- Rung 1-When NO (normally open) input 0000 becomes true DIFU 1000 becomes true.
- Rung 2- NO 1000 is true, NO 1001 remains false, NC 1001 remains true, NC 1000 turns false. Since we have a true path, (NO 1000 & NC 1001) OUT 1001 becomes true.
- Rung 3- NO 1001 is true therefore OUT 500 turns true.

Next Scan

- Rung 1- NO 0000 remains true. DIFU 1000 now becomes false. This is because the DIFU instruction is only true for one scan. (i.e. the rising edge of the logic before it on the rung)
- Rung 2- NO 1000 is false, NO 1001 remains true, NC 1001 is false, NC 1000 turns true. Since we STILL have a true path, (NO 1001 & NC 1000) OUT 1001 remains true.
- Rung 3- NO 1001 is true therefore OUT 500 remains true.

After 100 scans, NO 0000 turns off (becomes false). The logic remains in the same state as "next scan" shown above. (difu doesn't react therefore the logic stays the same on rungs 2 and 3)

On scan 101 NO 0000 turns back on. (becomes true)

- Rung 1-When NO (normally open) input 0000 becomes true DIFU 1000 becomes true.
- Rung 2- NO 1000 is true, NO 1001 remains true, NC 1001 becomes false, NC 1000 also becomes false. Since we no longer have a true path, OUT 1001 becomes false.
- Rung 3- NO 1001 is false therefore OUT 500 becomes false.

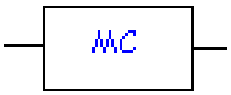
Master Controls

- Let's now look at what are called master controls. Master controls can be thought of as "*emergency stop switches*". An emergency stop switch typically is a big red button on a machine that will shut it off in cases of emergency. Next time you're at the local gas station look near the door on the outside to see an example of an e-stop.

  *IMPORTANT- We're not implying that this instruction is a substitute for a "hard wired" e-stop switch. There is no substitute for such a switch! Rather it's just an easy way to get to understand them.

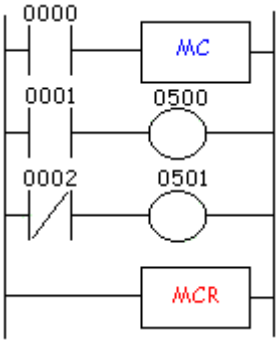
- The master control instruction typically is used in pairs with a master control reset. However this varies by manufacturer. Some use MCR in pairs instead of teaming it with another symbol. It is commonly abbreviated as MC/MCR (master control/master control reset), MCS/MCR (master control set/master control reset) or just simply MCR (master control reset).

Here is one example of how a master control symbol looks.



Below is an example of a master control reset.

- To make things interesting, many manufacturers make them act differently. Let's now take a look at how it's used in a ladder diagram. Consider the following example:



- Here's how different PLCs will run this program:
- **Manufacturer X**- In this example, rungs 2 and 3 are only executed when input 0000 is on (true). If input 0000 is not true the plc pretends that the logic between the mc and mcr instructions does not exist. It would therefore bypass this block of instructions and immediately go to the rung after the mcr instruction.
- Conversely, if input 0000 is true, the plc would execute rungs 2 and 3 and update the status of outputs 0500 and 0501 accordingly. So, if input 0000 is true, program execution goes to rung 2. If input 0001 is true 0500 will be true and hence it will turn on when the plc updates the outputs. If input 0002 is true (i.e. physically off) 0501 will be true and therefore it will turn on when the plc updates the outputs.
- **MCR just tells the plc "that's the end of the mc/mcr block".**
- In this plc, scan time is not extended when the mc/mcr block is not executed because the plc pretends the logic in the block doesn't exist. In other words, the instructions inside the block aren't seen by the plc and therefore it doesn't execute them.
- **Manufacturer Y**- In this example, rungs 2 and 3 are always executed regardless of the status of input 0000. If input 0000 is not true the plc executes the MC instruction. (i.e. MC becomes true) It then forces all the input instructions inside the block to be off. If input 0000 is true the MC instruction is made to be false.
- Then, if input 0000 is true, program execution goes to rung 2. If input 0001 is true 0500 will be true and hence it will turn on when the plc updates the outputs. If input 0002 is true (i.e. physically off) 0501 will be true and therefore it will turn on when the plc updates the outputs. MCR just tells the plc "that's the end of the mc/mcr block". When input 0000 is false, inputs 0001 and 0002 are forced off regardless if they're physically on or off. Therefore, outputs 0500 and 0501 will be false.
- The difference between manufacturers X and Y above is that in the Y scheme the scan time will be the same (well close to the same) regardless if the block is on or off. This is because the plc sees each instruction whether the block is on or off.
- Most all manufacturers will make a previously latched instruction (one that's inside the mc/mcr block) retain its previous condition.
- **If it was true before, it will remain true.**
  If it was false before, it will remain false.
- **Timers** should not be used inside the mc/mcr block because some manufacturers will reset them to zero when the block is false whereas other manufacturers will have them retain the current time state.
- **Counters** typically retain their current counted value.
- Here's the part to note most of all. When the mc/mcr block is off, (i.e. input 0000 would be false in the ladder example shown previously) an OUTB (OutBar or OutNot) instruction would not be physically on. It is forced physically off.

OutBar instruction

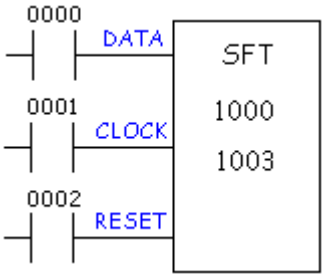
- In summary, BE CAREFUL! Most manufacturers use the manufacturer Y execution scheme shown above. When in doubt, however, read the manufacturers instruction manual. Better yet, just ask them.

Shift Registers

In many applications it is necessary to store the status of an event that has previously happened. As we've seen in past chapters this is a simple process. But what do we do if we must store many previous events and act upon them later.
Answer: we call upon the shift register instruction.

We use a register or group of registers to form a train of bits (cars) to store the previous on/off status. Each new change in status gets stored in the first bit and the remaining bits get shifted down the train. Huh? Read on.

The shift register goes by many names. SFT (ShiFT), BSL (Bit Shift Left), SFR (Shift Forward Register) are some of the common names. These registers shift the bits to the left. BSR (Bit Shift Right) and SFRN (Shift Forward Register Not) are some examples of instructions that shift bits to the right. We should note that not all manufacturers have shift registers that shift data to the right but most all do have left shifting registers.



A typical shift register instruction has a symbol like that shown above. Notice that the symbol needs 3 inputs and has some data inside the symbol.

The reasons for each input are as follows:

- **Data**- The data input gathers the true/false statuses that will be shifted down the train. When the data input is true the first bit (car) in the register (train) will be a 1. This data is only entered into the register (train) on the rising edge of the clock input.
- **Clock**- The clock input tells the shift register to "*do its thing*". On the rising edge of this input, the shift register shifts the data one location over inside the register and enters the status of the data input into the first bit. On each rising edge of this input the process will repeat.
- **Reset**- The reset input does just what it says. It clears all the bits inside the register we're using to 0.

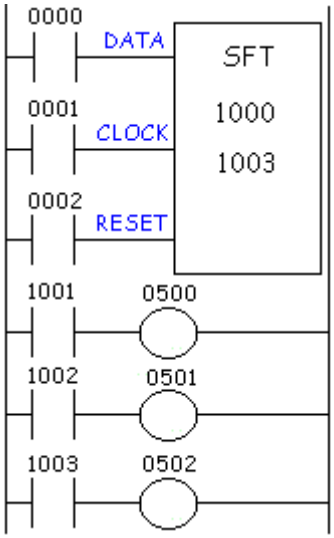
The 1000 inside the shift register symbol is the location of the first bit of our shift register. If we think of the shift register as a train (a choo-choo train that is) then this bit is the locomotive. The 1003 inside the symbol above is the last bit of our shift register. It is the caboose. Therefore, we can say that 1001 and 1002 are cars in between the locomotive and the caboose. They are intermediate bits. So, this shift register has 4 bits.(i.e. 1000,1001,1002,1003)

Lets examine an application to see why/how we can use the shift register.

Imagine an ice-cream cone machine. We have 4 steps. First we verify the cone is not broken. Next we put ice cream inside the cone.(turn on output 500) Next we add peanuts.(turn on output 501) And finally we add sprinkles.(turn on output 502) If the cone is broken we obviously don't want to add ice cream and the other items. Therefore we have to track the bad cone down our process line so that we can tell the machine not to add each item. We use a sensor to look at the bottom of the cone. (input 0000) If its on then the cone is perfect and if its off then the cone is broken. An encoder tracks the cone going down the conveyor. (input

0001) A push button on the machine will clear the register. (input 0002)

Here's what the ladder would look like:



Let's now follow the shift register as the operation takes place. Here's what the 1000 series register (the register we're shifting) looks like initially:

10xx Register															
15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
												0	0	0	0

A good cone comes in front of the sensor (input 0000). The sensor (data input) turns on. 1000 will not turn on until the rising edge of the encoder (input 0001). Finally the encoder now generates a pulse and the status of the data input (cone sensor input 0000) is transferred to bit 1000. The register now looks like:

10xx Register															
15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
												0	0	0	1

As the conveying system moves on, another cone comes in front of the sensor. This time it's a broken cone and the sensor remains off. Now the encoder generates another pulse. The old status of bit 1000 is transferred to bit 1001. The old status of 1001 shifts to 1002. The old status of 1002 shifts to 1003. And the new status of the data input (cone sensor) is transferred to bit 1000. The register now looks like:

10xx Register															
15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
												0	0	1	0

Since the register shows that 1001 is now on, the ladder says that output 0500 will turn on and ice cream is put in the cone.

As the conveying system continues to move on, another cone comes in front of the sensor. This time it's a good cone and the sensor turns on. Now the encoder generates another pulse. The old status of bit 1000 is transferred to bit 1001. The old status of 1001 shifts to 1002. The old status of 1002 shifts to 1003. And the new status of the data input (cone sensor) is transferred to bit 1000. The register now looks like:

10xx Register															
15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00

												0	1	0	1

Since the register shows that 1002 is now on the ladder says that output 0501 will turn on and peanuts are put on the cone. Since 1001 now holds the status of a broken cone, 500 remains off in the ladder above and no ice-cream is inserted into this cone. As the conveying system continues to move on, another cone comes in front of the sensor. This time it's also a good cone and the sensor turns on. Now the encoder generates another pulse. The old status of bit 1000 is transferred to bit 1001. The old status of 1001 shifts to 1002. The old status of 1002 shifts to 1003. And the new status of the data input (cone sensor) is transferred to bit 1000. The register now looks like:

10xx Register															
15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
												1	0	1	1

Since the register shows that 1003 is now on the ladder says that output 0502 will turn on and sprinkles are put on the cone. (Its done, yummy...)Since 1002 now holds the status of a broken cone, 501 remains off in the ladder above and no peanuts are put onto this cone. Since the register shows that 1001 is now on the ladder says that output 0500 will turn on and ice cream is put in that cone.

As the conveying system continues to move on, another cone comes in front of the sensor. This time it's another broken cone and the sensor turns off. Now the encoder generates another pulse. The old status of bit 1000 is transferred to bit 1001. The old status of 1001 shifts to 1002. The old status of 1002 shifts to 1003. And the new status of the data input (cone sensor) is transferred to bit 1000. The register now looks like:

10xx Register															
15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
												0	1	1	0

Notice that the status of our first cone has disappeared. In reality its sitting in location 1004 but it's useless for us to draw an application with 16 processes here. Suffice it to say that after the bit is shifted all the way to the left it disappears and is never seen again. In other words, it has been shifted out of the register and is erased from memory. Although it's not drawn, the operation above would continue on with each bit shifting on the rising edge of the encoder signal.

The shift register is most commonly used in conveyor systems, labeling or bottling applications, etc. Sometimes it's also conveniently used when the operation must be delayed in a fast moving bottling line. For example, a solenoid can't immediately kick out a bad can of beer when the sensor says its bad. By the time the solenoid would react the can would have already passed by. So typically the solenoid is located further down the conveyor line and a shift register tracks the can to be kicked out later when it's more convenient.

A shift register is often very difficult to understand. When in doubt, re-read the above and you'll understand it soon enough.

Getting and Moving Data

Let's now start working with some data. This is what can be considered to be getting into the "*advanced*" functions of a plc. This is also the point where we'll see some marked differences between many of the manufacturers functionality and implementation. On the lines that follow we'll explore two of the most popular ways to get and manipulate data.

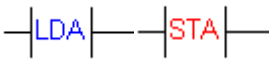
Why do we want to get or acquire data? The answer is simple. Let's say that we are using one of the manufacturers optional modules. Perhaps it's an A/D module. This module acquires Analog signals from the outside world (a varying voltage or current) and converts the signal to something the plc can understand (a digital signal i.e. 1's and 0's). Manufacturers automatically store this data into memory locations for us. However, we have to get the data out of there and move it some place else otherwise the next analog sample will replace the one we just took. In other words, move it or lose it! Something else we might want to do is store a constant (i.e. fancy word for a number), get some binary data off the input terminals (maybe a thumbwheel switch is connected there, for example), do some math and store the result in a different location, etc...

As was stated before there are typically 2 common instruction "*sets*" to accomplish this. Some manufacturers use a single instruction to do the entire operation while others use two separate instructions. The two are used together to accomplish the final result. Let's now look briefly at each instruction.

The single instruction is commonly called MOV (move). Some vendors also include a MOVN (move not). It has the same function of MOV but it transfers the data in inverted form. (i.e. if the bit was a 1, a 0 is stored/moved or if the bit was a 0, a 1 is stored/moved). The MOV typically looks like that shown below.

MOV instruction symbol

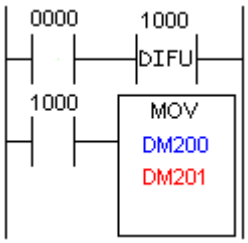
The paired instruction typically is called LDA (LoaD Accumulator) and STA (STore Accumulator). The accumulator is simply a register inside the CPU where the plc stores data temporarily while its working. The LDA instruction typically looks like that shown below, while the STA instruction looks like that shown below to the right.



Regardless of whether we use the one symbol or two symbol instruction set (we have no choice as it depends on whose plc we use) they work the same way.

Let's see the single instruction first. The MOV instruction needs to know 2 things from us.

- **Source (xxxx)**- This is where the data we want to move is located. We could write a constant here (2222 for example). This would mean our source data is the number 2222. We could also write a location or address of where the data we want to move is located. If we wrote DM100 this would move the data that is located in data memory 100.
- **Destination (yyyy)**- This is the location where the data will be moved to. We write an address here. For example if we write DM201 here the data would be moved into data memory 201. We could also write 0500 here. This would mean that the data would be moved to the physical outputs. 0500 would have the least significant bit, 0501 would have the next bit... 0515 would have the most significant bit. This would be useful if we had a binary display connected to the outputs and we wanted to display the value inside a counter for the machine operator at all times (for example).



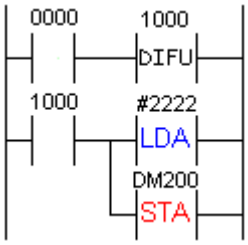
The ladder diagram to do this would look similar to that shown above.

Notice that we are also using a "difu" instruction here. The reason is simply because if we didn't the data would be moved during each and every scan. Sometimes this is a good thing (for example if we are acquiring data from an A/D module) but other times it's not (for example an external display would be unreadable because the data changes too much).

The ladder shows that each time real world input 0000 becomes true, difu will become true for only one scan. At this time LoaD 1000 will be true and the plc will move the data from data memory 200 and put it into data memory 201.
Simple but effective. If, instead of DM200, we had written 2222 in the symbol we would have moved (written) the number (constant) 2222 into DM201.

The two symbol instruction works in the same method but looks different. To use them we must also supply two things, one for each instruction:

- **LDA**- this instruction is similar to the source of a MOV instruction. This is where the data we want to move is located. We could write a constant here (2222 for example). This would mean our source data is the number 2222. We could also write a location or address of where the data we want to move is located. If we wrote DM100 this would move the data that is located in data memory 100.
- **STA**- this instruction is similar to the destination of a MOV instruction. We write an address here. For example if we write DM201 here the data would be moved into data memory 201. We could also write 0500 here. This would mean that the data would be moved to the physical outputs. 0500 would have the least significant bit, 0501 would have the next bit... 0515 would have the most significant bit. This would be useful if we had a binary display connected to the outputs and we wanted to display the value inside a counter for the machine operator at all times (for example).



The ladder diagram to do this would look similar to that shown above. Here again we notice that we are using a one-shot so that the move only occurs once for each time input 0000 becomes true. In this ladder we are moving the constant 2222 into data memory 200. The "#" is used by some manufactures to symbolize a decimal number. If we just used 2222 this plc would think it meant address 2222. PLCs are all the same... but they are all different.

Math Instructions

Let's now look at using some basic math functions on our data. Many times in our applications we must execute some type of mathematical formula on our data. It's a rare occurrence when our data is actually *exactly* what we needed.

As an example, let's say we are manufacturing widgets. We don't want to display the total number we've made today, but rather we want to display how many more we need to make today to meet our quota. Lets say our quota for today is 1000 pieces. We'll say X is our current production. Therefore, we can figure that 1000-X=widgets left to make. To implement this formula we obviously need some math capability.

In general, PLCs almost always include these math functions:

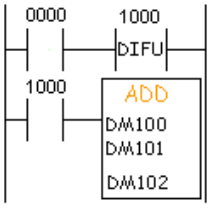
- **Addition**- The capability to add one piece of data to another. It is commonly called ADD.
- **Subtraction**- The capability to subtract one piece of data from another. It is commonly called SUB.
- **Multiplication**- The capability to multiply one piece of data by another. It is commonly called MUL.
- **Division**- The capability to divide one piece of data from another. It is commonly called DIV.

As we saw with the MOV instruction there are generally two common methods used by the majority of plc makers. The first method includes a single instruction that asks us for a few key pieces of information. This method typically requires:

- **Source A**- This is the address of the first piece of data we will use in our formula. In other words it's the location in memory of where the first "number" is that we use in the formula.
- **Source B**- This is the address of the second piece of data we will use in our formula. In other words it's the location in memory of where the second "number" is that we use in the formula. -NOTE: typically we can only work with 2 pieces of data at a time. In other words we can't work directly with a formula like 1+2+3. We would have to break it up into pieces. Like 1+2=X then X+3= our result.
- **Destination**- This is the address where the result of our formula will be put. For example, if 1+2=3, (I hope it still does!), the 3 would automatically be put into this destination memory location.

ADD symbol

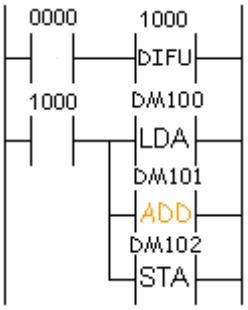
The instructions above typically have a symbol that looks like that shown above. Of course, the word ADD would be replaced by SUB, MUL, DIV, etc. In this symbol, The source A is DM100, the source B is DM101 and the destination is DM102. Therefore, the formula is simply whatever value is in DM100 + whatever value is in DM101. The result is automatically stored into DM102.



Shown above is how to use math functions on a ladder diagram. Please note that once again we are using a one-shot instruction. As we've seen before, this is because if we didn't use it we would execute the formula on every scan. Odds are good that we'd only want to execute the function one time when input 0000 becomes true. If we had previously put the number 100 into DM100 and 200 into DM101, the number 300 would be stored in DM102.(i.e. 100+200=300, right??)

ADD symbol (dual method)

The dual instruction method would use a symbol similar to that shown above. In this method, we give this symbol only the Source B location. The Source A location is given by the LDA instruction. The Destination would be included in the STA instruction.



Shown above is a ladder diagram showing what we mean.

The results are the same as the single instruction method shown above.

What would happen if we had a result that was greater than the value that could be stored in a memory location?

Typically the memory locations are 16-bit locations. (more about number types in a later chapter) In plain words this means that if the number is greater than 65535 (2^16=65536) it is too big to fit. Then we get what's called an overflow. Typically the plc turns on an internal relay that tells us an overflow has happened. Depending on the plc, we would have different data in the destination location. (DM102 from example) Most PLCs put the remainder here.

Some use 32-bit math which solves the problem. (except for really big numbers!) If we're doing division, for example, and we divide by zero (illegal) the overflow bit typically turns on as well. Suffice it to say, check the overflow bit in your ladder and if its true, plan appropriately.

Many PLCs also include other math capabilities. Some of these functions could include:

- **Square roots**
- **Scaling**
- **Absolute value**
- **Sine**
- **Cosine**
- **Tangent**
- **Natural logarithm**
- **Base 10 logarithm**
- **X^Y (X to the power of Y)**
- **Arcsine (tan, cos)**
- *and more....check with the manufacturer to be sure.*

Some PLCs can use floating point math as well. Floating point math is simply using decimal points. In other words, we could say that 10 divided by 3 is 3.333333 (floating point). Or we could say that 10 divided by 3 is 3 with a remainder of 1(long division). Many micro/mini PLCs don't include floating point math. Most larger systems typically do.