

CONTROLADORES LÓGICOS PROGRAMÁVEIS

PARTE II – SOFTWARE



1 INTRODUÇÃO

Qualquer sistema microprocessado precisa ser programado para funcionar adequadamente. Estes programas são armazenados na memória do sistema na forma de uma seqüência de códigos binários, representando as instruções do programa, que somente o processador tem capacidade para entender. Estas instruções codificadas são chamadas *código de máquina*. Para facilitar o trabalho dos programadores destes sistemas, cada fabricante de processador criou *códigos mnemônicos* para representar o conjunto de instruções entendidas pelos seus processadores. Como exemplo o mnemônico **LD** pode indicar a operação de armazenamento (carga - *load*) de um valor qualquer em um registrador interno do processador. A linguagem de programação que utiliza diretamente os códigos mnemônicos de um processador é chamada *linguagem Assembly*. Pelo fato da linguagem *Assembly* utilizar diretamente as instruções básicas de um processador, ela é considerada uma linguagem de *baixo nível*.

Mas a tarefa de programar sistemas com processadores pode ser mais fácil para os seres humanos com o uso das linguagens de *alto nível*. As linguagens de alto nível são linguagens de programação de processadores que utilizam comandos e instruções muito próximas das linguagens humanas, e portanto, mais fáceis de serem aprendidas e memorizadas. Como exemplo destas linguagens tem-se as linguagens C, C++, C#, JAVA, BASIC e outras.

Entretanto, quando os CLPs foram especificados, um dos requisitos básicos é que a forma de programação do controlador deveria ser simples e de fácil entendimento pelo pessoal de campo responsável para instalação e manutenção. Como as linguagens de programação convencionais não atendiam este requisito, foi necessário criar uma linguagem de programação baseada nos diagramas lógicos de contatos elétricos de relés. Esta linguagem de programação foi chamada *linguagem Ladder*.

O programa em linguagem Ladder era editado utilizando-se um terminal de programação, o qual tinha um teclado em que as teclas mostravam símbolos de contato (normalmente aberto / normalmente fechado), bobinas e ramificações. A Ilustração 1 mostra um diagrama lógico usando estes símbolos, com os quais os eletricitas de manutenção e os engenheiros eletricitas estão bem familiarizados e o programa equivalente na linguagem Ladder.

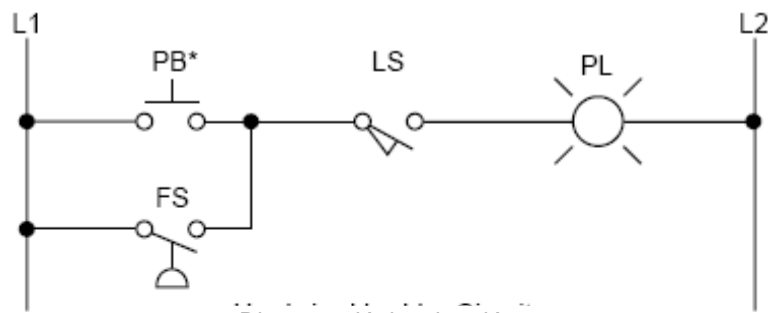
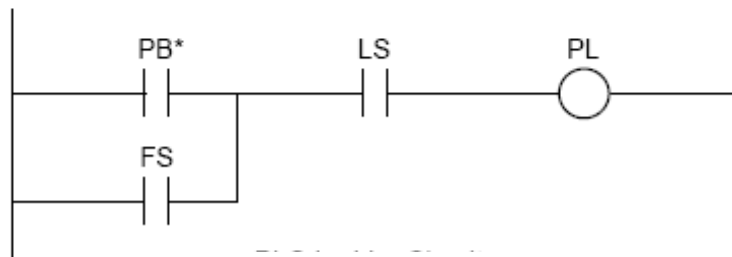


Diagrama lógico de relés



Programa Ladder equivalente

Ilustração 1: Exemplo de um diagrama lógico de contatos e um programa na linguagem Ladder

O terminal de programação compilava o diagrama Ladder para o código de máquina do processador, o qual era posteriormente carregado no CLP para execução. Durante a execução do programa, o terminal de programação permitia a visualização do programa de controle, animando os contatos e bobinas energizados para facilitar a análise do programa e depuração de erros.

Este método de escrever programas foi adotado pela maioria dos fabricantes de CLP. Entretanto, a medida que cada fabricante incluía recursos adicionais em seus CLPs, eles tendiam a criar elementos novos na linguagem Ladder, mas sem seguir nenhum padrão. Assim, cada fabricante acabava tendo sua linguagem Ladder própria, e normalmente incompatível com a linguagem dos demais fabricantes. Este fato criou a necessidade de se adotar uma norma internacional para padronizar os métodos, linguagens e ferramentas de programação de CLPs. Esta norma, publicada em 1993 pela IEC, é a norma IEC 61131-3.

2 NORMA IEC 61131-3

A norma IEC 61131 é um conjunto de documentos que estabelece padrões de compatibilidade entre CLPs de qualquer fabricante em sistemas de automação baseados nestes equipamentos.

A terceira parte desta norma (IEC 61131-3) estabelece padrões para a sintaxe e componentes específicos das linguagens de programação para CLPs. Ela detalha vários métodos de programação que devem ser empregados, os tipos de dados que devem estar disponíveis. Além disto, cria um modelo de *software* e estabelece padrões para a estruturação de programas em CLPs, definindo também funções pré-definidas que tem que ser disponibilizadas pelas ferramentas de *software* fornecidas pelos fabricantes de CLPs.

2.1 LINGUAGENS E INSTRUÇÕES

A norma IEC 61131-3 define duas linguagens gráficas e duas linguagens baseadas em texto para a programação de CLPs. As linguagens gráficas usam símbolos para representar as instruções do programa de controle, enquanto as linguagens baseadas em texto usam palavras como instruções.

As linguagens gráficas são:

- Diagramas Ladder (LD)
- Diagramas de Blocos Funcionais (FBD)

As linguagens baseadas em texto são:

- Lista de Instruções (IL)
- Texto Estruturado (ST)

Adicionalmente, a norma IEC61131-3 inclui uma estrutura que permite a programação orientada a objetos, este recurso é chamado Gráfico Sequencial de Funções (SFC). O SFC é muitas vezes considerado como uma linguagem de programação da norma IEC 61131-3, mas realmente ele é uma ferramenta que permite criar programas de forma organizada usando as quatro linguagens de programação definidas pela norma (LD, FBD, IL e ST).

A estrutura do SFC é muito semelhante aos fluxogramas utilizados em programação de computadores. Nele, tarefas de controle diferentes podem ser escritas utilizando linguagens diferentes. A estrutura básica do SFC é baseada no método de modelagem de sistemas GRAFCET (IEC 848).

A norma IEC 61131-3 estabelece um método de programação em blocos gráfico ou orientado a objetos, este método aumenta a flexibilidade na criação e depuração de programas para os CLPs. O método permite que seções de um programa sejam agrupadas individualmente como *tasks*, as quais podem ser facilmente interligadas com o restante do programa. Assim, um programa completo, que siga a norma IEC 61131-3, pode ser formado por diversas pequenas *tasks*, representando um bloco gráfico dentro de um SFC. Pela norma IEC 61131-3, é possível a combinação de linguagens durante a criação de um programa de controle no CLP. Este recurso facilita as tarefas de programação e depuração nos CLPs, já que torna possível implementar cada solução de controle através da linguagem de programação mais apropriada.

A norma IEC 61131-3 define uma grande variedade de funções e blocos funcionais e todos operam com *variáveis*. Cada variável em um programa deve poder armazenar um determinado *tipo de dado*, e a norma define um grande número de tipos de dados. A Tabela 1 mostra alguns exemplos dos tipos de dados, funções e blocos funcionais definidos pela norma.

Os tipos de dados estão relacionados com o tipo de informação recebida pelo CLP, tais como informações binárias, números reais, informações de data e hora. Já as funções são operações que manipulam estes dados, tais como comparação, inversão e adição. Os blocos funcionais são conjuntos de funções, arranjadas na forma de instruções, criados para trabalhar com blocos de dados. As variáveis, além de estarem associadas a um tipo de dado, devem também ser associadas a um *escopo*. O escopo de uma variável se refere a característica de utilizar a variável em uma aplicação. As *variáveis globais* podem ser usadas por qualquer programa na aplicação, enquanto as *variáveis locais* podem ser usadas somente por um programa específico.

A norma IEC 61131-3, além de definir tipos de variáveis, funções e blocos funcionais padronizados, permite que usuários e fabricantes de CLPs definam seus próprios elementos de programação. Assim, a norma não especifica um conjunto fixo e rígido de elementos para programação, mas estabelece um conjunto básico de elementos

padronizados, e permite o acréscimo de elementos adicionais conforme as características do CLP e da aplicação em que ele será utilizado.

Tabela 1: Elementos de programação definidos pela norma IEC 61131-3

Tipos de dados de variáveis	<ul style="list-style-type: none"> • <i>Strings</i> baseadas em <i>bits</i> (booleanos ou <i>bit</i>, <i>byte</i>, <i>word</i>) • Números inteiros (com ou sem sinal) • Números reais • Hora e data • <i>Strings</i> de caracteres ASCII • Definidos pelo usuário ou pelo fabricante do CLP (simples ou matriz)
Funções	<ul style="list-style-type: none"> • Baseadas em <i>bits</i> (booleanas: AND, OR, NOT, etc.) • Numéricas/aritméticas (ADD, SUB, MUL, DIV, SQR, LOG, LN, SIN, COS, TAN, etc.) • Funções de conversão de dados • Funções seletivas (LIMIT, MAX, MIN, etc.) • Comparações (>, <, =, >=, <=, <>) • Funções de <i>strings</i> ASCII (LEN, LEFT, RIGHT, INSERT, REPLACE, DELETE, etc.) • Funções definidas pelo usuário ou pelo fabricante do CLP
Blocos funcionais	<ul style="list-style-type: none"> • <i>Flip-flops</i> • Detectores de bordas (↑, ↓) • Contadores (crescente, decrescente, crescente/decrescente) • Temporizadores (TON, TOF) • Blocos definidos pelo usuário ou pelo fabricante do CLP
Escopo de variáveis	<ul style="list-style-type: none"> • Global • Local

A flexibilidade em relação a tipos de dados e funções da norma IEC 61131-3 permite aos fabricantes de CLPs fornecerem instruções que considerem necessárias, as que não estejam definidas pela norma. Tais instruções podem servir a recursos muito específicos, como por exemplo, um bloco funcional de controle de posicionamento para operar um módulo de controle de servo-motor. Este bloco funcional pode utilizar como parâmetros variáveis com os tipos de dados definidos pela norma, mas este bloco somente estará disponível para ser usado em programas dos CLPs compatíveis com o módulo. Outros CLPs, mesmo em conformidade com a norma, não poderão utilizar tal bloco funcional.

2.2 DECLARAÇÃO DE VARIÁVEIS

Durante a implementação de um programa de controle em um CLP, o usuário deve nomear, ou declarar, as variáveis usadas pelo programa. Esta declaração de variáveis não é nada mais que mapear os endereços físicos das entradas e saídas do CLP, indicando qual dispositivo de campo está conectado a qual entrada ou saída do CLP. A Ilustração 2(a) mostra uma chave fim de curso (LS1) ligada a uma entrada de um CLP. Nesta configuração, o dispositivo é identificado pelo programa de controle por seu endereço (010). Entretanto, pela norma IEC 61131-3, o dispositivo pode ser identificado através de um nome alfanumérico, composto por letras, dígitos e o caractere subscrito (_). Assim, a chave fim de curso pode ser declarada como uma variável chamada **Limit_Switch_1**, ou qualquer outro nome apropriado, como na Ilustração 2(b). A partir do momento que uma variável é declarada, ela será conhecida pelo mesmo nome por todo o programa de controle, independente da linguagem de programação IEC 61131-3 usada. O nome associado à variável não é sensível a caso, ou seja, não existe distinção entre letras maiúsculas e minúsculas.

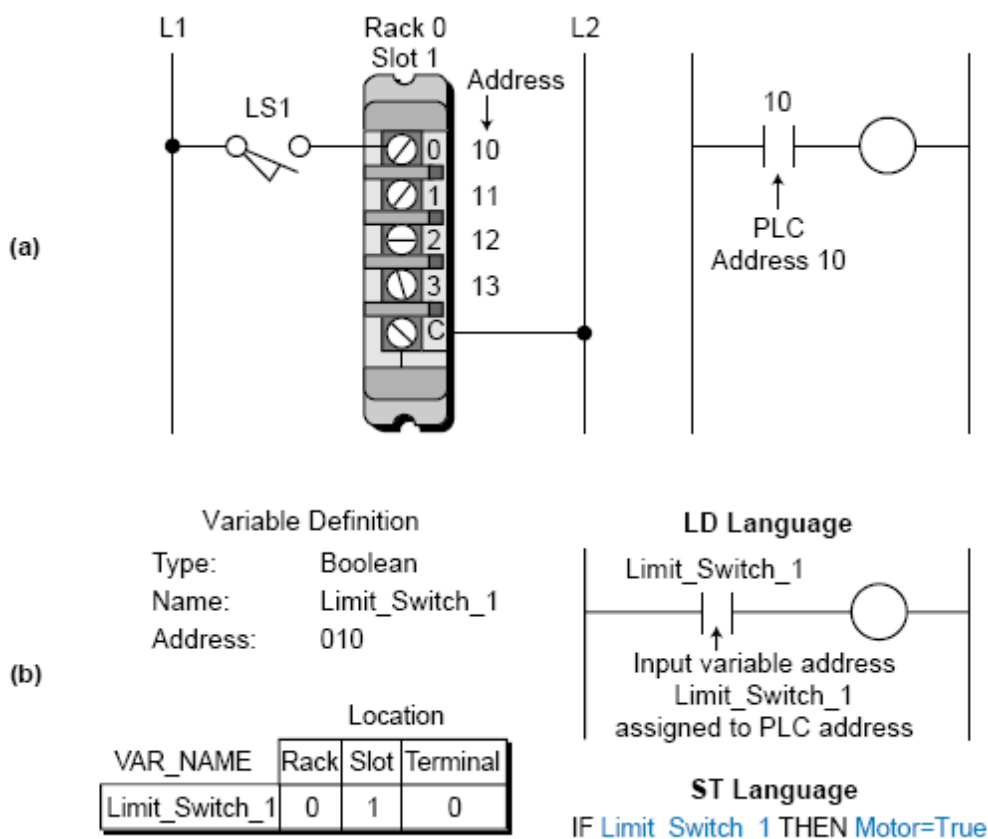


Ilustração 2: Ligação de um dispositivo e sua identificação em um CLP, conforme a norma IEC 61131-3

Tabela 2: Tipos de dados para declaração de variáveis

Classe	Tipo de Variável	Descrição	Exemplo
Discreto	Booleano	TRUE (1) FALSE (0)	Chaves fim de curso, motores, botoeiras
Analógico	Inteiro	-128 +3764	Valores de temporizadores, valores de contadores, resultados de cálculos inteiros
	Real (ponto flutuante)	-34.573 +1.35x10 ³	E/S analógica, cálculos gerais
String ASCII	Strings de mensagens	"Temperatura"	Mostrar informações em um monitor ou impressora
Sistema	Interno	Relés de controle (Booleanos)	"Bobinas" e "contatos" de relés internos, saídas de temporizadores
		Variáveis inteiras Variáveis reais	Cálculos aritméticos internos
	Entrada	Variáveis conectadas às interfaces de entrada	Entradas discretas ou analógicas
	Saída	Variáveis conectadas às interfaces de saída	Saídas discretas ou analógicas

Quando declara uma variável, o usuário deve especificar o tipo da variável, além do seu nome. Isto permite que o CLP conheça que tipo de informação o dispositivo irá transmitir para a variável correspondente. A norma IEC 61131-3 suporta muitos tipos de dados para variáveis locais e globais, como pode ser visto na Tabela 2. Entretanto, os três tipos mais comuns são:

- Booleano
- Inteiro
- Real

As variáveis booleanas são variáveis que ocupam um único *bit*, significando que os dados transmitidos e recebidos por elas estão na forma de 0s e 1s. As variáveis de E/S discretas pertencem a esta categoria, então elas devem ser especificadas como variáveis “Bool” no programa de controle. Muitas variáveis não discretas, tais como os sinais analógicos de entrada que são lidos através de um cartão de entrada analógico, são variáveis inteiras, porque elas transmitem a informação na forma de números inteiros (202, -127). Assim, elas devem ser especificadas no programa de controle como variáveis inteiras. Variáveis internas que transmitam informação na forma de frações ou de números em ponto flutuante (2.7×10^2) são variáveis reais.

2.3 LINGUAGENS DE PROGRAMAÇÃO

Apesar da norma de programação IEC 61131-3 oferecer diversos recursos avançados de programação para os usuários de CLPs, ela está realmente baseada na lógica dos diagramas lógicos de relés (Ladder) devido a simplicidade de uso e de grande penetração entre os engenheiros e técnicos de manutenção da área elétrica. Entretanto, a norma reduz a necessidade de intertravamentos de circuitos complexos, muito comuns nos diagramas lógicos de relés.

A seguir será feita uma descrição detalhada de cada uma das linguagens de programação definidas pela norma IEC 61131-3. É importante notar que, quando se programar um CLP que respeite a norma IEC 61131-3, qualquer uma das linguagens pode ser utilizada, tanto de forma única, como agrupadas. Quando usadas de forma agrupada, normalmente utiliza-se o SFC para estruturar as seqüências lógicas do programa.

2.3.1 DIAGRAMAS LADDER (LD)

Os CLPs foram desenvolvidos para serem facilmente programados, usando os símbolos e expressões dos diagramas lógicos de relés (Ladder) para representar o programa de controle de máquinas ou processos. A linguagem de programação resultante desta idéia, a qual usa os símbolos básicos dos diagramas lógicos de relés, foi chamada de *linguagem Ladder*.

Funcionamento básico

A linguagem Ladder é um conjunto de instruções representadas por símbolos gráficos. Estes símbolos são ligados de tal forma a se obter uma lógica de controle, a qual é armazenada na memória de aplicação do CLP.

A funcionalidade principal um programa em linguagem Ladder é que controlar as saídas de um CLP através da análise lógica de suas entradas. Os diagramas Ladder usam *degraus* (*rungs*) para atingir este objetivo. A Ilustração 3 mostra a estrutura básica de um *rung*. Geralmente um *rung* consiste em um conjunto de condições de entrada, representadas por instruções de contatos, e uma instrução de saída no final do *rung*, esta representada por um símbolo de bobina (*coil*).

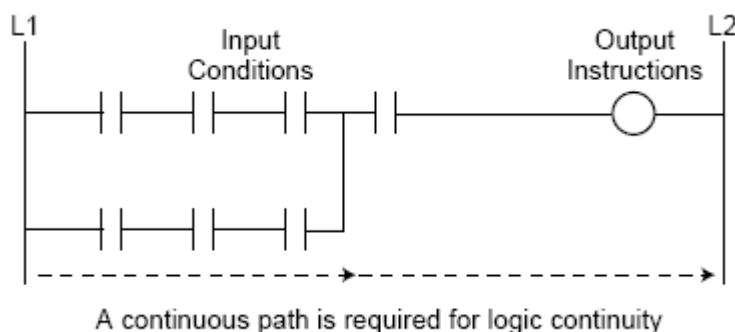


Ilustração 3: Estrutura básica de um rung na linguagem Ladder

Um *rung* é considerado ativo (TRUE), ou seja, apresenta a sua saída energizada, quando ele apresenta *continuidade lógica*. A continuidade lógica existe quando existir o fluxo de corrente através do *rung*, da esquerda para a direita. A execução de eventos lógicos que habilitam a saída é que determina esta continuidade. Em um *rung* da linguagem Ladder, a linha vertical mais à esquerda simula uma linha de energia com um potencial elétrico positivo, enquanto a linha vertical mais à direita simula uma linha de energia com potencial elétrico negativo. A continuidade ocorre quando existe um caminho fechado que permite a circulação de corrente entre as linhas de energia. A Ilustração 4 mostra diversos caminhos, para um mesmo *rung*, que oferecem continuidade. Pode-se notar que em cada caminho, as condições dos contatos de entrada são diferentes.

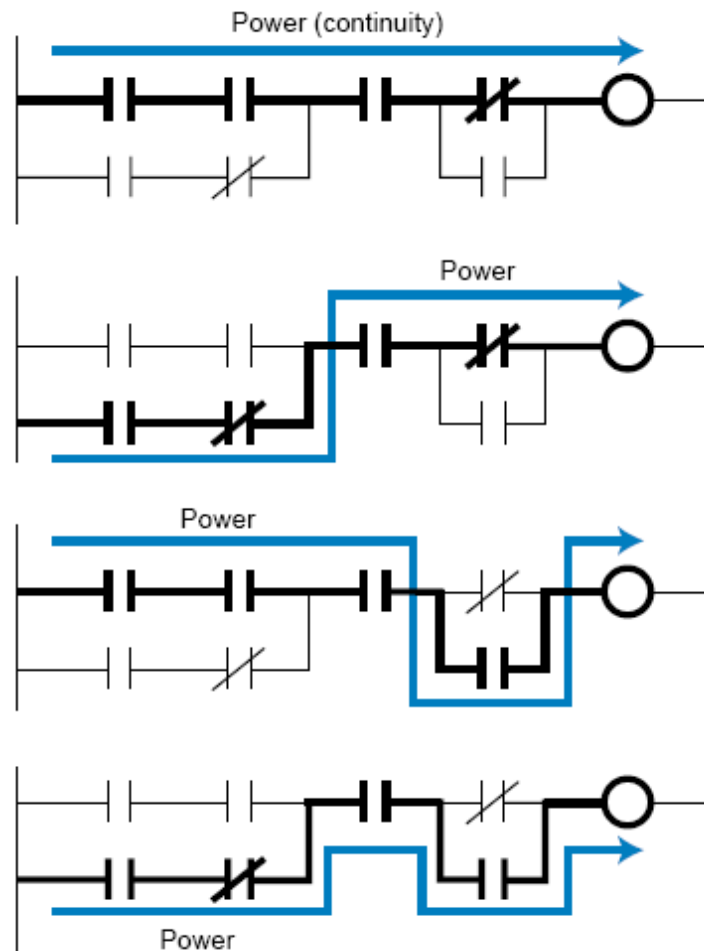


Ilustração 4: Exemplos de continuidade lógica em um rung

Uso de blocos funcionais

A evolução, ao longo dos anos, da linguagem Ladder original adicionou a ela um poderoso conjunto de novas instruções. Estas novas instruções são normalmente representadas através de blocos funcionais. O uso de blocos funcionais aumenta a flexibilidade e o poder de programação da linguagem Ladder básica. A Ilustração 5 mostra o uso de blocos funcionais, juntamente com símbolos básicos da linguagem Ladder.

Quando um programa Ladder contém um bloco funcional, as instruções de contato são usadas para representar as condições de entrada que selecionam ou habilitam o bloco. Um bloco funcional pode ter um ou mais entradas que controlam sua operação. Além disso, os blocos funcionais podem ter uma ou mais bobinas de saída, que representam o estado da função que está sendo executada pelo bloco.

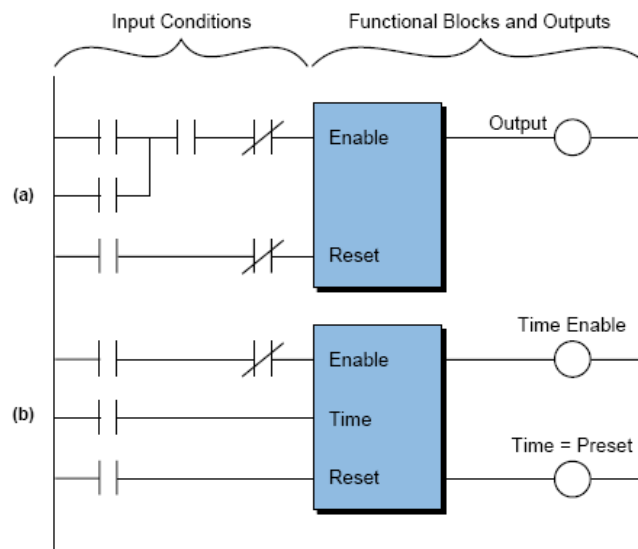


Ilustração 5: Exemplos do uso de blocos funcionais na linguagem Ladder

Caminho reverso

Uma regra, presente em quase todos os CLPs, previne o caminho reverso (da direita para a esquerda) do fluxo de energia em um *rung*. A lógica dos CLPs não permite o fluxo reverso de energia para evitar caminhos falsos. Um caminho falso ocorre quando o fluxo de energia segue por uma direção inversa indesejada, através de algum dispositivo de campo. Assim, uma continuidade não verdadeira ocorre. Se uma lógica de controle implementada em CLP necessitar de um caminho reverso, o usuário deve reprogramar o *rung*, para que haja somente caminhos diretos. A Ilustração 6 mostra um exemplo de *rung* com caminho reverso.

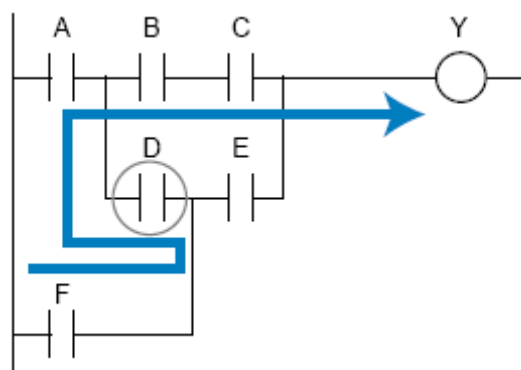


Ilustração 6: Exemplo de rung com caminho reverso

Instruções de relés

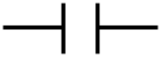





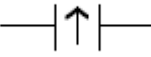
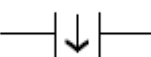
As instruções de relés são as instruções mais básicas dentro do conjunto de instruções da linguagem Ladder. Estas instruções representam o estado ON/OFF das entradas e saídas do CLP. As instruções de relés usam dois tipos de símbolos: contatos e

bobinas. Os contatos representam as condições de entrada que devem ser avaliadas, em um determinado *rung*, para controlar uma saída. As bobinas representam uma saída de um *rung*. A Tabela 3 lista as instruções de relés mais comuns.

Em um programa, cada contato e bobina tem um endereço numérico, o qual identifica o que está sendo avaliado e o que está sendo controlado. Este endereço se refere a localização na tabela de E/S onde os dispositivos físicos de E/S estão conectados ao CLP, ou a *bits internos* utilizados pelo programa de controle. Um contato, independente se representa uma conexão de entrada ou saída ou um *bit interno*, pode ser utilizado por todo o programa de controle, sempre que a condição que ele representa precisar ser avaliada.

O formato dos contatos de um *rung* em um programa de CLP depende da lógica de controle desejada. Os contatos podem ser colocados em qualquer configuração série, paralelo ou série/paralelo necessária para controlar uma determinada saída. Quando a continuidade lógica existir em pelo menos um caminho de contatos, sempre da esquerda para a direita, a condição do *rung* é considerada verdadeira (TRUE), e as saídas controladas pelo *rung* são ativadas. A condição do *rung* é considerada falsa (FALSE) quando não existir nenhum caminho com continuidade lógica, então as saídas controladas pelo *rung* são desativadas.

Tabela 3: Instruções básicas de relés da linguagem Ladder

Instruções de Relés		
Instrução	Símbolo	Função
Avaliar contato normalmente aberto		Testa por uma condição ON em um determinado endereço
Avaliar contato normalmente fechado		Testa por uma condição OFF em um determinado endereço
Ativar bobina de saída		Liga uma saída ou um <i>bit interno</i> quando a lógica do <i>rung</i> for verdadeira (TRUE)
Desativar bobina de saída		Desliga uma saída ou um <i>bit interno</i> quando a lógica do <i>rung</i> for verdadeira (TRUE)
Ativar bobina de saída com retenção (SET)		Liga uma saída ou um <i>bit interno</i> quando a lógica do <i>rung</i> for verdadeira (TRUE), e mantém assim mesmo quando a lógica do <i>rung</i> tornar-se falsa (FALSE)
Desativar bobina de saída com retenção (RESET)		Desliga uma bobina ativada pela instrução SET quando a lógica do <i>rung</i> for verdadeira (TRUE)
Detecção de bordas subida		Contato que fecha por um ciclo de varredura quando a entrada referenciada sofrer uma transição positiva (0 → 1)
		Contato que fecha por um ciclo de varredura quando a entrada referenciada sofrer uma transição negativa (1 → 0)

Estas instruções de relés são as instruções mais básicas da linguagem Ladder. Com estas instruções é possível criar programas de controle que examine os estados ON/OFF de *bits* endereçados na memória do CLP e controle o estado de saídas internas e externas.

Avaliação de contatos normalmente abertos

Uma instrução de avaliação de contatos normalmente abertos (NA) testa a condição ON em um endereço de referência. Este endereço de referência pode ser um *bit* da tabela de entradas, representando um dispositivo ligado a uma entrada física do CLP;

um *bit interno*, armazenado na área de armazenamento da tabela de dados; ou um *bit* da tabela de saídas, representando um dispositivo ligado a uma saída física do CLP.

Durante a execução da instrução de avaliação de contato NA no programa de controle, o processador examina o endereço de referência por uma condição ON. Se o valor do *bit* no endereço de referência for 0 (OFF), o processador considera que não existe continuidade lógica por aquele contato do *rung*, conforme a Ilustração 7(a). Entretanto, se o valor do *bit* no endereço de referência for 1 (ON), o processador considera que existe continuidade lógica por aquele contato do *rung*, conforme a Ilustração 7(b).

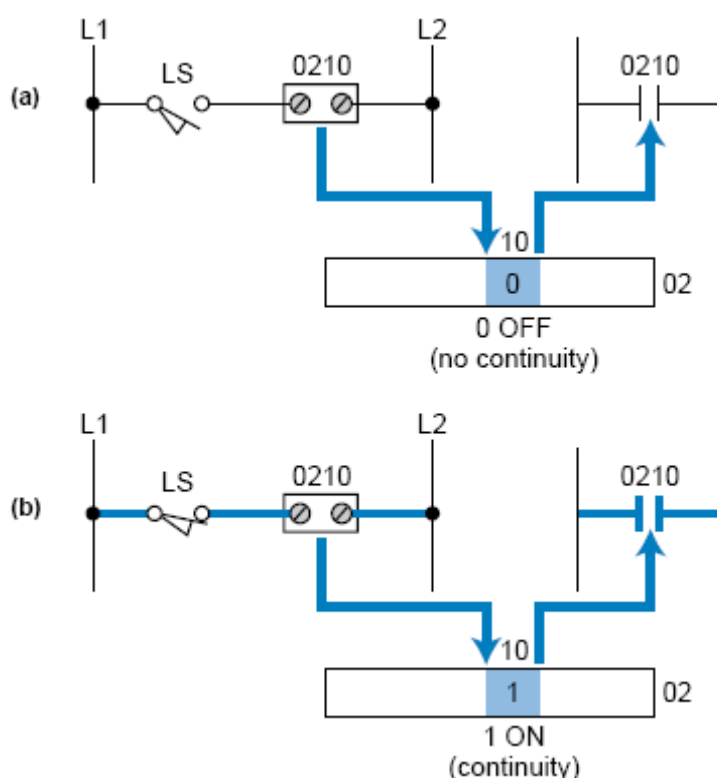


Ilustração 7: Exemplo da execução da instrução de avaliação de contatos NA

Avaliação de contatos normalmente fechados

Uma instrução de avaliação de contatos normalmente fechados (NF) testa a condição OFF em um endereço de referência. Da mesma forma que na instrução de avaliação de contato NA, os endereços de referência nesta instrução podem ser *bits* da tabela de entradas, da tabela de saídas ou um *bit interno*.

Durante a execução da instrução de avaliação de contato NF no programa de controle, o processador examina o endereço de referência por uma condição OFF. Se o valor do *bit* no endereço de referência for 0 (OFF), o processador considera que existe

continuidade lógica por aquele contato do *rung*, conforme a Ilustração 8(a). Entretanto, se o valor do *bit* no endereço de referência for 1 (ON), o processador considera que não existe continuidade lógica por aquele contato do *rung*, conforme a Ilustração 8(b). Esta instrução pode ser associada a função lógica NOT, ou seja, se o endereço de referência não está ligado (NOT ON), a continuidade lógica existe.

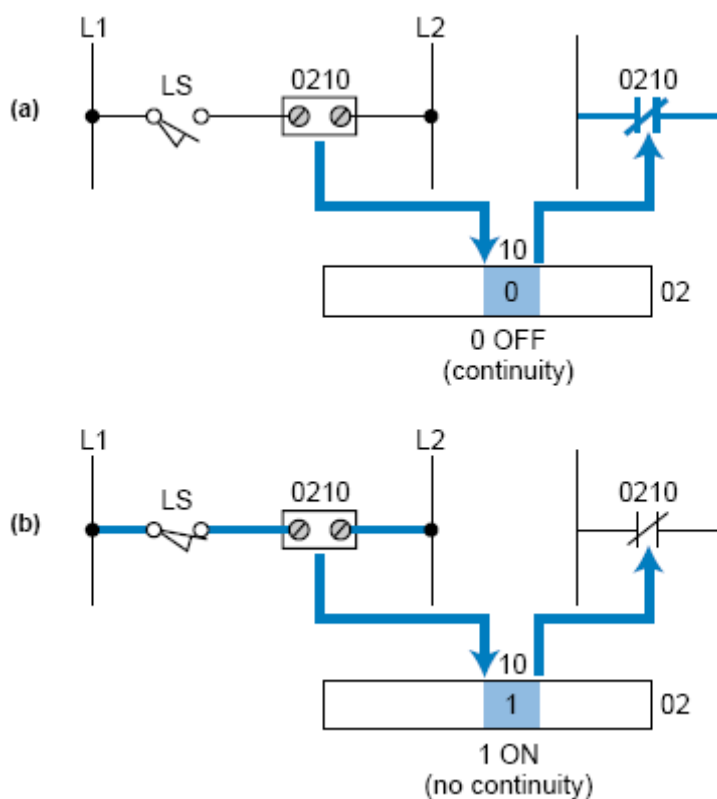


Ilustração 8: Exemplo da execução da instrução de avaliação de contatos NF

Acionamento de bobinas de saída

Uma instrução de acionamento de bobina de saída controla tanto uma saída física (um dispositivo atuador ligado à uma interface de saída do CLP) como uma saída interna (um *bit interno*). Esta instrução usa um *bit* de bobina endereçado na área de armazenamento ou na tabela de saídas como endereço de referência. O símbolo —()— também pode ser usado para representar uma instrução de acionamento de bobina.

Durante a execução de uma instrução de acionamento de bobina, o processador avalia todas as condições de entrada (contatos) no *rung*. Se não existir a continuidade lógica no *rung*, o processador escreve o valor 0 (OFF) no *bit* do endereço de referência da instrução, conforme a Ilustração 9(a). Isso indica que a saída deverá ser desligada quando da varredura de atualização de saídas, no caso de uma saída física. Agora, se o

processador detectar uma continuidade lógica por qualquer caminho em um *rung*, ele escreve o valor 1 (ON) no *bit* do endereço de referência da instrução, conforme a Ilustração 9(b). Este valor 1 no endereço de referência, caso este represente uma saída física, faz com que a saída correspondente seja ligada durante a varredura de atualização de saídas. É importante lembrar que esta varredura ocorre somente quando todos os *rungs* do programa de controle forem avaliados.

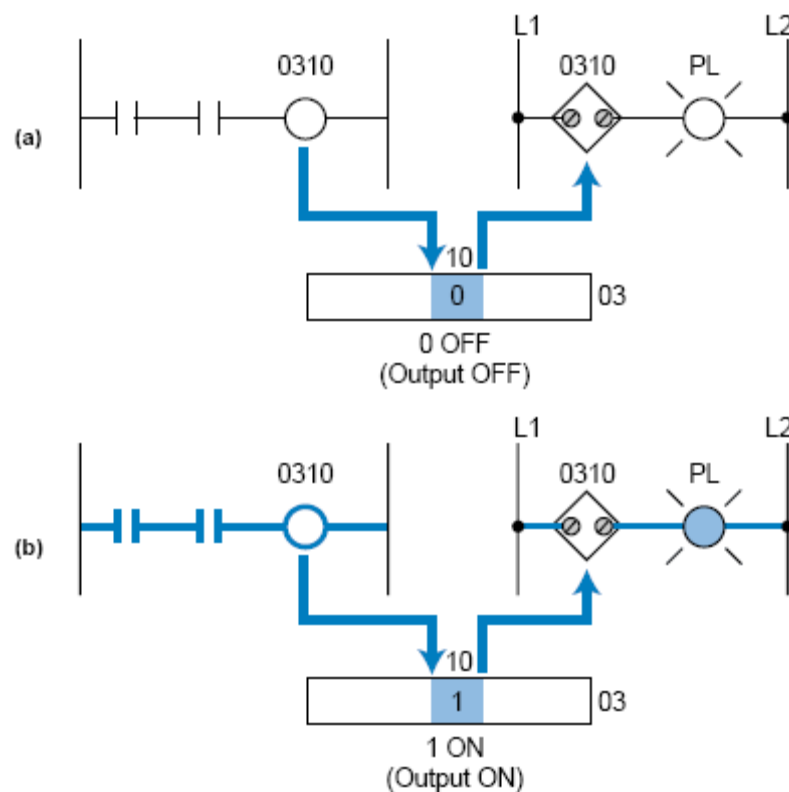


Ilustração 9: Exemplo da execução de uma instrução de acionamento de bobina de saída

Quando a bobina de saída endereçada é um *bit interno*, quando a bobina é acionada, o *bit interno* na memória de armazenamento recebe o valor lógico 1. E quando a bobina é desacionada, o *bit interno* recebe o valor lógico 0. Estes *bits internos* são usados quando o programa de controle necessita de seqüências de intertravamento ou quando saídas físicas não são necessárias.

Desligamento de bobinas de saída

Uma instrução de desligamento de bobina de saída é, essencialmente, a instrução oposta a de acionamento de bobina de saída. Se não existir continuidade nos contatos de um *rung*, a instrução faz com que o *bit* da saída endereçada receba o valor lógico 1 (ON). Assim sendo, quando existir continuidade lógica no *rung*, a instrução

escreve o valor lógico 0 (OFF) no *bit* da saída endereçada. O símbolo —(/)— também representa a instrução de desligamento de bobina de saída em alguns CLPs.

Operações de bobinas de saída com retenção (SET/RESET)

Existem duas instruções que podem operar de forma retentiva sobre as bobinas de saída, a instrução de acionamento e desligamento das bobinas, normalmente chamadas por SET e RESET.

Uma instrução de acionamento de bobina de saída com retenção (SET) tem a característica de ligar uma saída, quando existir uma continuidade lógica no *rung*, e mantê-la ligada mesmo quando cessar a continuidade lógica do *rung* que a acionou. A saída permanecerá ligada até que uma instrução de desligamento de bobina de saída com retenção (RESET) for executada pelo programa de controle.

Uma instrução de desligamento de bobina de saída com retenção (RESET) é a única forma de desligar uma bobina de saída acionada por uma operação SET. Quando existir no *rung* algum caminho com continuidade lógica, esta instrução desliga a bobina do endereço de referência. A Ilustração 10 mostra o uso das instruções SET e RESET.

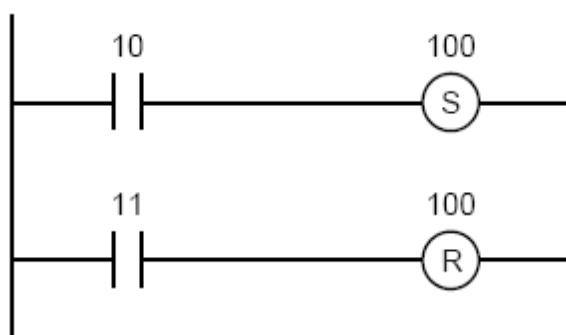


Ilustração 10: Exemplo do uso das instruções SET e RESET

Deteção de bordas

As instruções de detecção de bordas fornecem uma condição de continuidade lógica temporária (pelo tempo de um único ciclo de varredura) ao contato associado quando ocorrer uma transição positiva ($0 \rightarrow 1$) ou uma transição negativa ($1 \rightarrow 0$) no *bit* do endereço de referência associado à instrução. O endereço de referência para estas instruções podem ser tanto entradas ou saídas físicas (*bits* das tabelas de entradas e saídas) como *bits internos*. A Ilustração 11 mostra dois exemplos utilizando estas instruções e como é o comportamento dos contatos destas instruções.

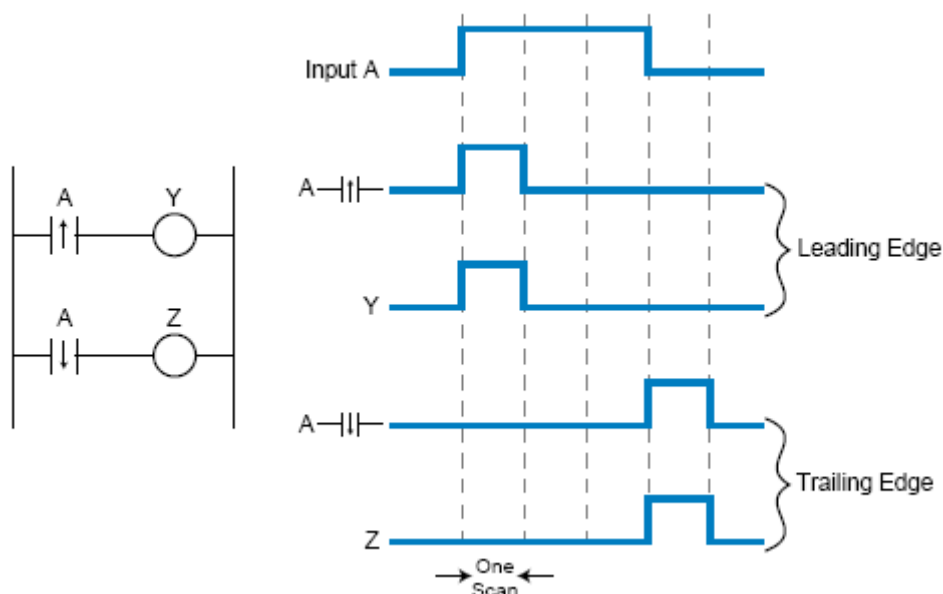


Ilustração 11: Comportamento das instruções de detecção de bordas

Varredura de avaliação de um programa Ladder

A varredura de avaliação é um conceito importante, já que ela define a ordem na qual o processador executa um programa Ladder. O processador começa a executar um programa Ladder logo após a leitura dos estados de todas as entradas e armazena estas informações na tabela de entradas (varredura de atualização das entradas). A execução começa pelo primeiro *rung* do programa, e prossegue, executando um *rung* de cada vez, de cima para baixo. A medida que o processador executa o programa de controle, ele examina o endereço de referência de cada instrução, assim ele pode avaliar a continuidade lógica para cada *rung*. Mesmo que as condições de saída de um *rung* avaliado afete as condições de *rungs* avaliados anteriormente, o processador não retornará a eles para reavaliá-los.

Para tornar isto claro, observe o diagrama da Ilustração 12, o qual apresenta quatro *rungs* simples. O contato NA, identificado pelo número 10, o qual se assume que corresponda a uma botoeira ligada a uma entrada física do CLP, ativa o primeiro *rung* quando a botoeira for pressionada. Se o contato 10 fechar, a saída 100 será ligada. No próximo *rung*, o contato de estado da saída 100 ligará a saída 101, o contato 101 ligará a saída 102 e por fim, o contato 102 ligará a saída 103. Mesmo estando em *rungs* diferentes, todas as saídas serão ligadas no mesmo ciclo de varredura, já que o programa executivo faz a varredura de atualização das saídas somente ao final da varredura do

programa de controle. Este comportamento está ilustrado na Ilustração 12.

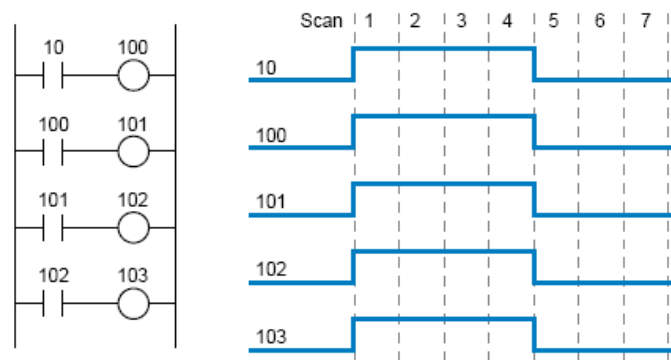


Ilustração 12: Comportamento do acionamento de saídas devido à varredura de atualização do programa de controle

Já a Ilustração 13 apresenta o mesmo programa de controle da Ilustração 12, mas com a ordem dos *rungs* invertida. Assumindo-se que a botoeira ligada à entrada 10 é pressionada no primeiro ciclo de varredura do CLP, o CLP somente terá as quatro saídas (100, 101, 102 e 103) acionadas após quatro ciclos completos de varredura.

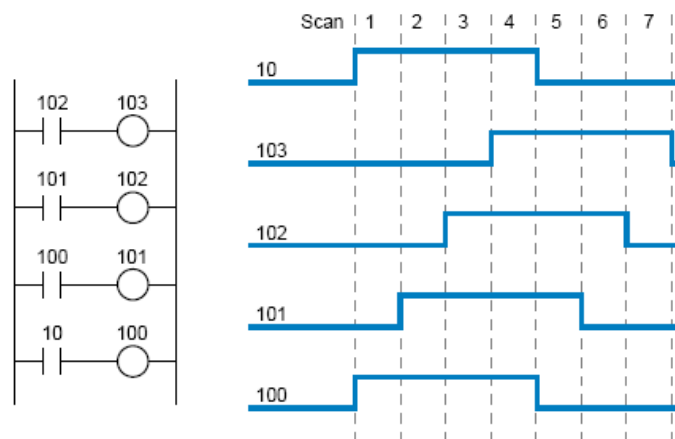


Ilustração 13: Comportamento do acionamento de saídas devido à varredura de atualização do programa de controle, com a ordem dos rungs invertida

A lógica que o processador usa no primeiro ciclo de varredura é a seguinte:

1. O processador detecta o nível lógico 1 na entrada 10 e escreve este valor no *bit* correspondente na tabela de entradas.
2. O processador começa o ciclo de avaliação do programa pelo primeiro *rung*, verificando que o contato 102 está desligado (OFF), e portanto, a saída 103 deve permanecer desligada (OFF).

3. No segundo *rung*, o contato 101 está desligado (OFF), então a saída 102 deve permanecer desligada (OFF).

4. No terceiro *rung*, o contato 100 está desligado (OFF), então a saída 101 deve permanecer desligada (OFF).

5. No quarto *rung*, o contato 10 está ligado (ON), assim a saída 100 deve ser ligada (ON).

6. Chegando ao final da avaliação do programa de controle, o processador realiza a varredura de atualização das saídas, ligando efetivamente a saída 100. O ciclo deve então recomençar.

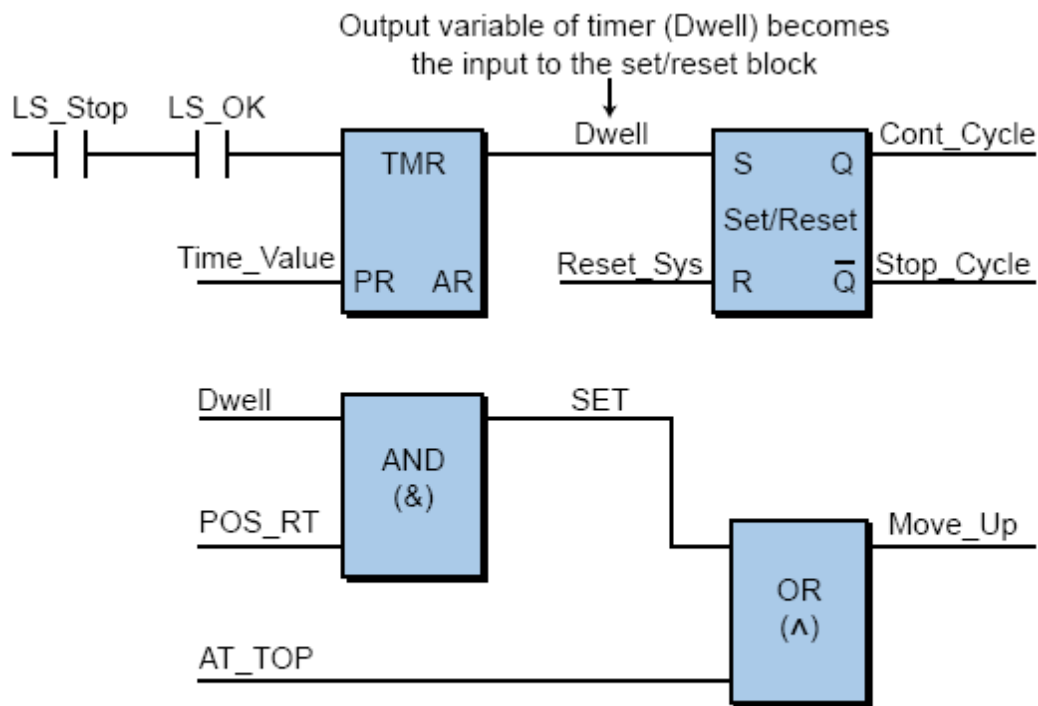
No próximo ciclo, estando a botoeira ainda pressionada, a saída 101 será ligada porque, no final do ciclo anterior, a saída 100 tinha sido ligada. E esta lógica seguirá até o final do quarto ciclo de varredura, quando todas as saídas estarão ligadas. As saídas serão desligadas obedecendo a mesma seqüência quando a botoeira 10 for liberada.

Um detalhe importante a se lembrar quando implementar um programa na linguagem Ladder é que, se for necessário que uma saída de um *rung* tenha efeito, como contato de entrada, em outro *rung*, este segundo deve sempre ser programado após o primeiro. Se isso não for respeitado, podem ocorrer problemas durante a execução do programa de controle, como mostrado nos exemplos das ilustrações 12 e 13.

2.3.2 DIAGRAMAS DE BLOCOS FUNCIONAIS (FBD)

O diagrama de blocos funcionais (FBD) é uma linguagem gráfica que permite ao usuário criar programas para CLPs interligando elementos gráficos como se fossem componentes elétricos. A Ilustração 14 mostra um exemplo de programa para CLP usando FBD.

A norma IEC 61131-3 estabelece a forma gráfica de como representar os blocos e as regras para a interligação entre eles. A norma também estabelece uma grande quantidade de blocos padrão, que devem ser disponibilizados por qualquer sistema que permita a programação FBD. Entretanto, a norma deixa livre aos fabricantes de CLPs disponibilizarem blocos não-padrão, que possam controlar recursos especiais disponíveis nos CLPs.



Section of a control program using a timer, set/reset, AND, and OR function blocks

Ilustração 14: Exemplo de um programa em FBD

Além dos blocos padrão e dos blocos específicos dos fabricantes, a norma IEC 61131-3 permite aos usuários criarem seus próprios blocos, de acordo com os requisitos dos programas de controle. Isto é conhecido como *encapsular um bloco funcional*. A vantagem em criar blocos funcionais é que eles podem ser construídos a partir da ligação de outros blocos funcionais, ou utilizando qualquer uma das outras linguagens de programação para CLPs, o que cria uma grande flexibilidade na programação por blocos funcionais. O encapsulamento permite que um usuário crie novos blocos funcionais, e os armazene em uma *biblioteca*, e os blocos desta biblioteca podem ser utilizados diversas vezes no mesmo programa de controle, ou até mesmo em outros programas de controle feitos por outros programadores.

Os blocos personalizados podem ser utilizados em conjunto com os blocos padrão ou com os blocos específicos dos fabricantes, podendo todos estes serem integrados em programas na linguagem FBD e na linguagem Ladder.

2.3.3 LISTA DE INSTRUÇÕES (IL)

A lista de instruções (IL) é uma linguagem de programação de baixo nível, similar a linguagem de máquina (linguagem *assembly*) usada pelos microprocessadores. Um programa de exemplo é mostrado na Ilustração 15. Este tipo de linguagem é útil para pequenas aplicações, bem como em aplicações que necessitam de otimização na velocidade de execução do programa ou de alguma rotina específica do programa. Como mencionado antes, com a linguagem IL é possível criar blocos funcionais.

Instruções		Comentários
LD	b1	(*resultado corrente:=TRUE*)
AND	b2	(*resultado corrente:=b1 AND b2*)
ANDN	b3	(*resultado corrente:=b1 AND b2 AND NOT b3*)
ST	b0	(*b0:=resultado corrente*)

Nota: O resultado corrente é mantido em um registrador de resultado. A última instrução armazena o valor do registrador de resultado como a variável b0.

Ilustração 15: Exemplo de um programa em IL

2.3.4 TEXTO ESTRUTURADO (ST)

O texto estruturado (ST) é uma linguagem de programação de alto nível, que permite programar CLPs utilizando o paradigma da programação estruturada. Por este paradigma, para resolver tarefas complexas é necessário dividi-las em inúmeras tarefas menores e mais simples. A linguagem ST lembra muito as linguagens de programação de computadores BASIC e PASCAL, como é mostrado na Ilustração 16. Estas linguagens, assim como a linguagem ST, usam *sub-rotinas* para executar diferentes partes de uma função de controle, e passam parâmetros e valores entre estas diferentes partes do programa.

Assim com as linguagens LD, FBD e IL, a linguagem ST utiliza a definição de variáveis para identificar os dispositivos de campo de entrada e saída, além de qualquer outra variável criada internamente e usada pelo programa.

```

IF Manual AND NOT Alarm THEN
    Level:=Manual_Level;
    Mixer:=Start AND NOT Reset
ELSE_IF Other_Mode THEN
    Level:=Max_Level;
ELSE Level:=(Level_Indic ´ 100)/Scale;
END_IF;

```

Ilustração 16: Exemplo de um programa em ST

Diferente das linguagens LD, IL e FBD, a linguagem ST apresenta comandos que permitem iterações, ou laços, tais como WHILE..DO e REPEAT..UNTIL. Ela também apresenta comandos para execuções condicionais, tais como IF..THEN..ELSE. Além disso, ela suporta operações aritméticas (+, -, *, /), booleanas (AND, OR, NOT) e uma variedade de funções para processamento de diversos tipos de dados, tais como informações de datas e horas.

A linguagem ST é extremamente útil para a execução de rotinas como geração de relatórios, onde as instruções baseadas na língua inglesa, explicam exatamente o que está sendo feito. Deve-se lembrar que a linguagem ST pode ser usada para encapsular, ou criar, um bloco funcional que realizará uma determinada tarefa disparada por uma lógica de controle, conforme a Ilustração 17. Este bloco funcional pode ser utilizado inúmeras vezes por todo o programa de controle.

A programação usando linguagem ST é particularmente conveniente nas aplicações que envolvem manipulação de dados, computação de ordenação e com uso intensivo de matemática com valores de ponto flutuante. A linguagem ST também é a melhor escolha para a implementação de controles com inteligência artificial (IA), lógica *fuzzy* e controles com tomadas de decisão.

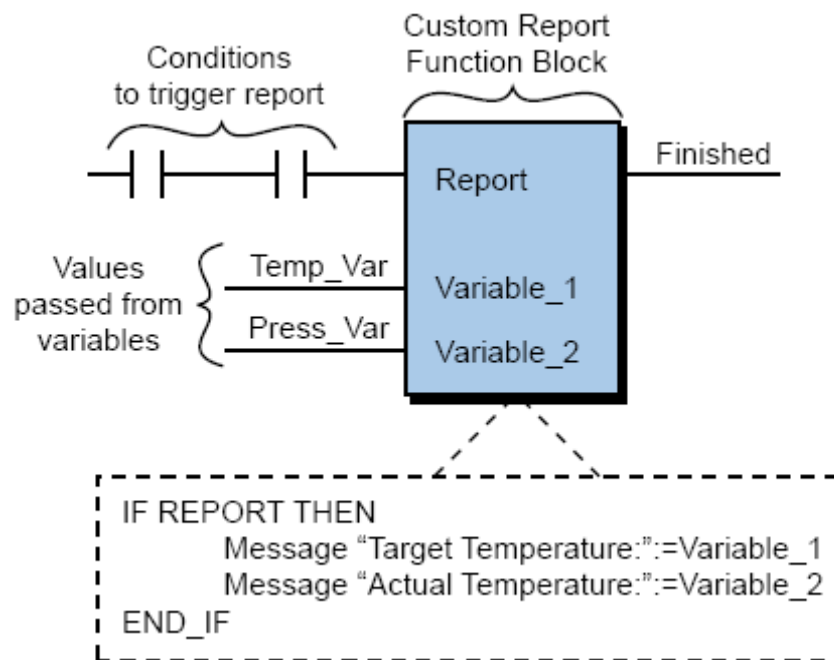


Ilustração 17: Bloco funcional escrito em ST e acionado por uma lógica LD

2.3.5 GRÁFICO SEQUENCIAL DE FUNÇÕES (SFC)

O gráfico sequencial de funções (SFC) é uma “linguagem” gráfica que fornece uma representação na forma de diagramas para seqüências de controle em um programa. Basicamente, o SFC é uma estrutura de suporte semelhante a um fluxograma que ajuda o programador a organizar os sub-programas, ou sub-rotinas (programadas em LD, FBD, IL ou ST) que formam o programa de controle. O SFC é particularmente útil em operações de controle sequencial, onde um programa executa diversos passos, um após outro, quando determinadas condições são satisfeitas.

A estrutura de programação SFC contém três elementos básicos que organizam o programa de controle:

- Etapas
- Transições
- Ações

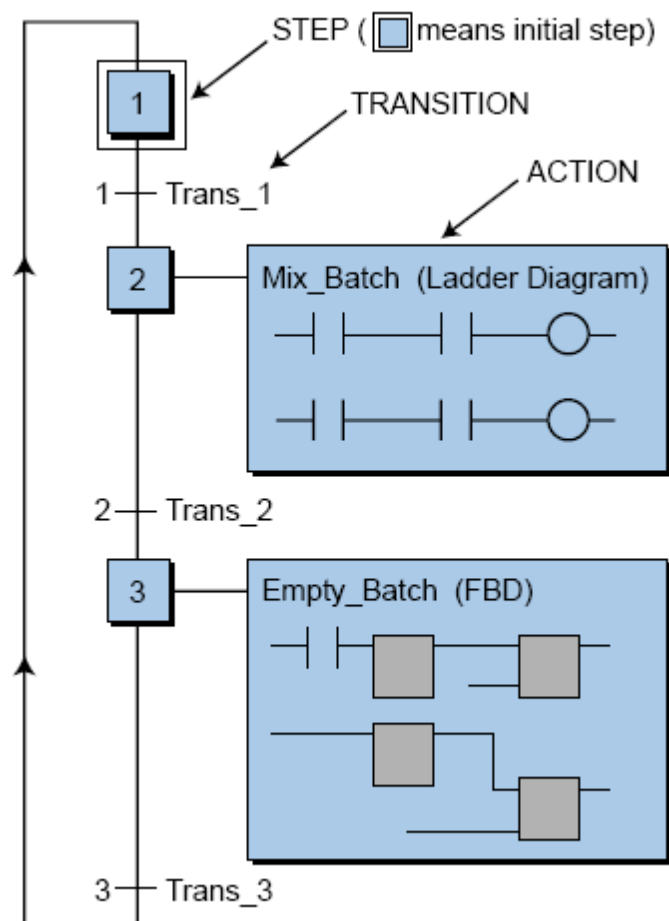


Ilustração 18: Exemplo de um SFC

Uma etapa é um estágio em um processo de controle. Por exemplo, a aplicação de mistura mostrada na Ilustração 18 apresenta três etapas: a etapa inicial, a etapa de mistura e a etapa de esvaziamento. Quando o programa de controle recebe uma entrada, ele executará cada etapa, começando pela etapa 1. Cada etapa pode ou não ter uma ação associada a ela. Uma ação é um conjunto de instruções de controle que fazem o CLP executar certas funções de controle durante a etapa. Uma ação pode ser programada usando qualquer uma das quatro linguagens da norma IEC 61131-3. Após o CLP executar uma etapa/ação, ele deve receber uma transição antes de prosseguir para a próxima etapa. Uma transição pode ocorrer na forma da atuação de uma variável de entrada, um resultado de uma ação anterior, ou um comando condicional IF (por exemplo, IF Temp_1 ≥ 100). Assim, para a aplicação mostrada na Ilustração 19, o CLP executará a ação 2 somente depois da etapa 1 receber uma entrada válida e a transição 1 ocorrer, isto é, a chave fim de curso LS_Reach disparar. Depois do CLP terminar a ação 2, ele irá esperar pela transição 2 (IF Temp_1 ≥ 100) ocorrer e então passar para a etapa 3.

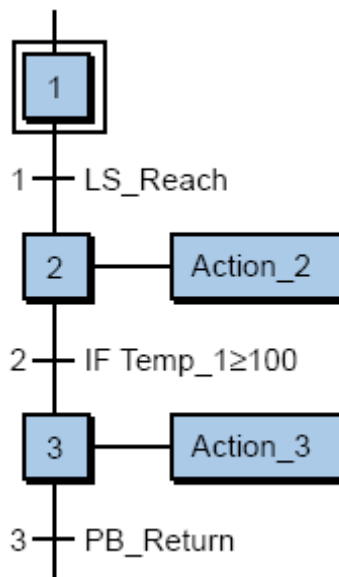


Ilustração 19: Transições e ações em um SFC

Como mencionado anteriormente, a linguagem SFC tem sua origem na norma francesa do Grafcet. O Grafcet também usa etapas, transições e ações, e opera da mesma maneira do SFC. No Grafcet, quando uma etapa está ativa, o processador varre a lógica de E/S pertinente às ações da etapa, bem como a lógica para a transição imediatamente posterior a etapa. Assim como no Grafcet, o SFC é similar a um fluxograma na forma como o controle é passado de uma etapa para outra. Assim como no Grafcet, o SFC pode ser programado para trabalhar diretamente com temporizações e diagramas de eventos.

A maior diferença entre o Grafcet e o SFC é que o Grafcet permite somente comandos de ações escritos, tais como “Abrir válvula”, para ligar ou desligar dispositivos. O SFC permite implementar ações de diversas maneiras, usando as linguagens LD, IL, ST e FBD, ou uma combinação entre elas, inclusive permitindo o uso de blocos funcionais personalizados.

Os SFCs podem ser imaginados como objetos de construção de blocos, usados para criar a estrutura básica de “todo” o programa de controle, enquanto as outras linguagens são usadas para implementar os detalhes “dentro” do SFC. De fato, os SFCs podem ter as chamadas *macro-etapas*, as quais permitem um SFC principal ter outro SFC como ação de uma etapa, conforme a Ilustração 20. Este SFC menor, embutido no maior, tem suas próprias etapas, transições e ações, e age de forma similar a uma sub-rotina em um programa de computador.

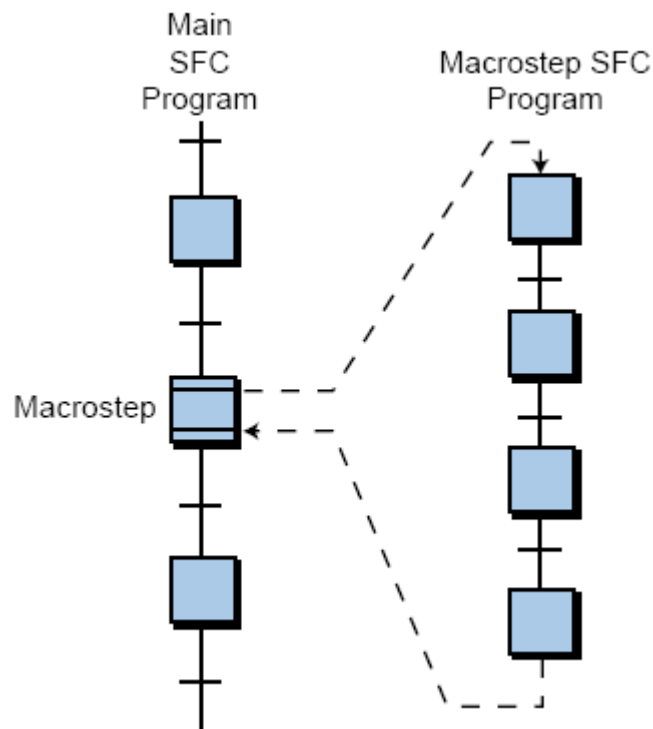


Ilustração 20: Uso de um SFC como uma macro-etapa