

Apostila de Automação Industrial

João Paulo Cerquinho Cajueiro

28 de abril de 2015

Sumário

1 Sistemas de automação	3
1.1 História	4
1.2 Classificação	5
1.3 Pirâmide de automação	6
2 Gerenciamento da manufatura: PIMS e MES	8
2.1 PIMS	8
2.1.1 Historiador de Processos	9
2.1.2 Banco de Dados Temporal	11
2.1.3 Interface Gráfica	11
2.2 MES	12
2.2.1 Redes industriais	16
2.3 Gerenciamento da manufatura: MES	19
2.4 Redes de Comunicação: Introdução e noções básicas	20
2.4.1 Modelo OSI	21
2.4.2 Internet	24
3 SCADA – Supervisory Control and Data Acquisition	26
3.1 Arquitetura de sistemas SCADA e interfaceamento com níveis de automação	26
3.2 Funcionalidades principais de sistemas SCADA	26
4 Sistema SCADA MANGO	27
4.1 Instalação	27
4.2 Uso	27
4.3 <i>Watch list</i>	29
4.4 Tela gráfica	29
5 Controladores Lógico-Programáveis (CLP)	35

6 Programação - arduino	36
6.1 Piscando um led.	37
6.2 Sinais analógicos	39
6.3 Controle: for e if	40
6.4 Comunicação com o computador	41
6.5 plcLib	43
6.5.1 Acumulador	44
6.5.2 Funções lógicas	45
6.5.3 Memória	47
6.5.4 Temporizadores	48
6.5.5 Contadores	49
6.6 Linguagem Ladder – Lógica Booleana	51
6.7 De relês a CLPs	52
6.7.1 Diagrama Ladder	53
6.8 Mapas de Karnaugh	56
6.8.1 Mapas de n variáveis	61
6.9 Equações e circuitos não-completamente especificados.	63
6.10 Linguagem Ladder – Temporizadores	65
6.11 Linguagem Ladder – Contadores	65
6.12 Linguagem Ladder – Aplicações	65
7 Linguagem Grafct	66
7.1 Linguagem Grafct – Aplicações Parte 1	71
7.2 Linguagem Grafct – Aplicações Parte 2	71
8 Instrumentação Industrial	72
8.1 Medição de grandezas mecânicas. Características de instrumentos. .	72
8.2 Transmissão de dados, aterramento e blindagem em instrumentação.	72

Capítulo 1

Sistemas de automação

A palavra automação vem do latim *automatus* – mover por si mesmo. Logo a automação de uma tarefa consiste em fazer esta tarefa ser realizada sem trabalho humano. Isto pode ser por diversos motivos: seja por que é uma tarefa perigosa e portanto queremos aumentar a segurança das pessoas, como num processo que envolva alta temperatura, por exemplo; seja para fazer a tarefa de forma mais rápida, seja para melhorar a qualidade do produto final ou seja porque simplesmente o custo do trabalho humano é muito elevado. Logo, podemos definir automação da seguinte forma:

Automação é a substituição do trabalho humano para melhorar segurança, qualidade, produção e custos.

Neste contexto, automação industrial é nada mais que a automação de um sistema industrial, ou de um sistema de manufatura. Embora manufatura venha de fazer com as mãos, a revolução industrial mudou seu conceito, passando a significar a fabricação de praticamente qualquer produto. Do ponto de vista econômico, a manufatura é a transformação de materiais (matéria prima) em itens de maior valor (produto). Isto é conseguido por uma determinada sequência de processos químicos e físicos. De forma mais sucinta:

Manufatura é a transformação de matéria prima em produtos pela aplicação de um ou mais processos.

Logo, a automação industrial consiste em fazer os processos necessários para a manufatura com o mínimo de esforço ou interferência humana, visando melhor segurança, qualidade, produção e custo. Note que por esforço humano, queremos dizer tanto esforço físico quanto mental, logo uma máquina bastante complexa mas que funcione a manivela, não se classificaria como automação; da mesma forma uma máquina que não exija esforço físico mas requer atenção constante também não é automatizada (seria apenas mecanizada).

1.1 História

É interessante pegar alguns pontos chaves na história da automação. Embora várias máquinas mecânicas de diversas graus de complexidade já existissem, como por exemplo relógios mecânicos desde o século VIII na China e desde o século XIII na Europa e os fantásticos robôs de Pierre Jaquet-Droz, do século XVIII, considera-se que a revolução industrial iniciou com a invenção do tear mecânico por Cartwright em 1785, que realiza um movimento relativamente complexo de forma automática a partir de uma roda d'água.

Um outro grande avanço ocorreu por volta de 1788, com a invenção do mecanismo de regulagem de fluxo de vapor de James Watt, o que permitia então controlar a potência de caldeiras e outras máquinas a vapor, controlando uma energia

muito maior que uma roda d'água. Isto foi um grande impulso para a mecanização, mas a verdadeira automatização ainda ficava muito restrita devido a dificuldade de realizar processos complexos de forma automática. Ou seja, retirava-se grande parte do esforço físico do homem, mas ainda era necessário muito esforço mental.

Em 1820 Babbage começou a desenvolver a sua máquina diferencial, que hoje chamaríamos de uma calculadora mecânica. Ela evoluiu até o conceito da máquina analítica, descrita em 1837, que é considerada o primeiro projeto de computador, embora apenas partes dela tenham sido efetivamente construídas.

Em 1880, Herman Hollerith criou um novo método baseado na utilização de cartões perfurados, para automatizar algumas tarefas de tabulação do censo dos EUA que antes duravam 10 anos. Com o método, o processo era concluído em 6.

Ao longo da primeira metade do século XX foram utilizados muitos sistemas eletromecânicos para o controle de processos industriais. Eram os chamados circuitos chaveados, que utilizavam relés para o controle lógico e para o comando de motores.

Em 1936, Alan Turing descreveu um *computador universal* em seu artigo “On Computable Numbers, with an Application to the Entscheidungsproblem”, o que hoje é conhecido como uma Máquina de Turing. Suas idéias foram desenvolvidas em 1944, com a construção do Colossus, considerado como o primeiro computador, embora tivesse a função específica de quebrar o código criptográfico alemão na segunda guerra. Ele consistia de um circuito com 1600-2400 válvulas com capacidade de processamento de 25k caracteres/s. A título de comparação, os computadores de casa de hoje em dia atingem 10 bilhões de cálculos por segundo.

O avanço da eletrônica fez que a capacidade de processamento dos computadores aumentasse de forma exponencial. Atualmente o mais rápido computador do mundo é o chinês Tianhe-2, que faz 33,86 quatrilhões de cálculos por segundo, consumindo 24MW.

Na década de 60 a General Motors fez uma especificação de um CLP – Controlador Lógico Programável, que é um computador voltado para o controle de processos industriais. Em 1968 foi criado o primeiro.

Hoje em dia toda automação está relacionada a sistemas computadorizados, seja em CLPs, CNC, robôs industriais, automação dos sistemas de apoio a produção, entre outros.

1.2 Classificação

Hoje em dia se definem, grosso modo, 3 tipos de automação, tal qual mostra a figura 1.1: fixa, flexível e programável.

A automação fixa é aplicada à produção de um único produto (ou com mínimas variações), em grandes quantidades: refinaria de petróleo, parafuso, tampas de

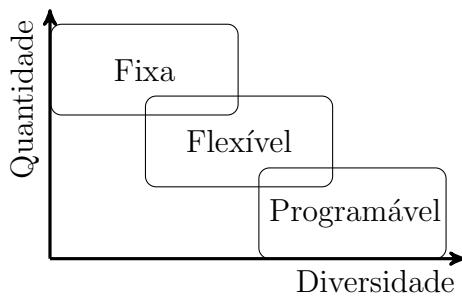


Figura 1.1: Tipos de automação industrial quanto a quantidade e diversidade de produtos.

garrafa, clips, biscoito, cerveja, etc. Ela utiliza equipamentos específicos para aquela tarefa, que portanto tem alto custo mas grande produtividade.

A automação flexível é aplicada à produção de produtos parecidos, em que pequenas modificações permitem a alteração do produto, como por exemplo mudança de um perfil a ser prensado ou extrudado ou a mudança das quantidades do mesmo conjunto de matérias primas (mudança de receita). Tipicamente é feita a chamada fabricação em lotes, onde entre um lote e outro se alteram as peças e/ou as sequências a serem seguidas de forma automática para ter o menor tempo parado possível. Exemplos são livros, circuitos integrados, potes de plástico, máquinas de café.

A automação programável é para produção de produtos diferentes mas cujo volume de produção não justifica um processo único. Ela usa máquinas de propósito geral, tais como robôs, ferramentas de controle numérico e impressoras 3d, onde a definição do processo é quase toda feita por *software*, de modo que o custo do maquinário é diluído em diversos produtos.

A tendência é cada vez mais ter a automação flexível e programável aumentando a capacidade de produção, de modo que a flexível vai ocupando nichos da fixa e a programável da flexível. Apesar disso, em vários casos é difícil imaginar alguns produtos deixando de utilizar a automação fixa.

1.3 Pirâmide de automação

A automação em larga escala de uma grande indústria ou de um conjunto de indústrias é mais complexa que a manufatura: envolve problemas de abastecimento, armazenagem, análise de mercado, exigências ambientais, entre várias outras coisas. Uma forma de se separar os diferentes problemas da automação é através da chamada Pirâmide de Automação, mostrada na figura 1.2.

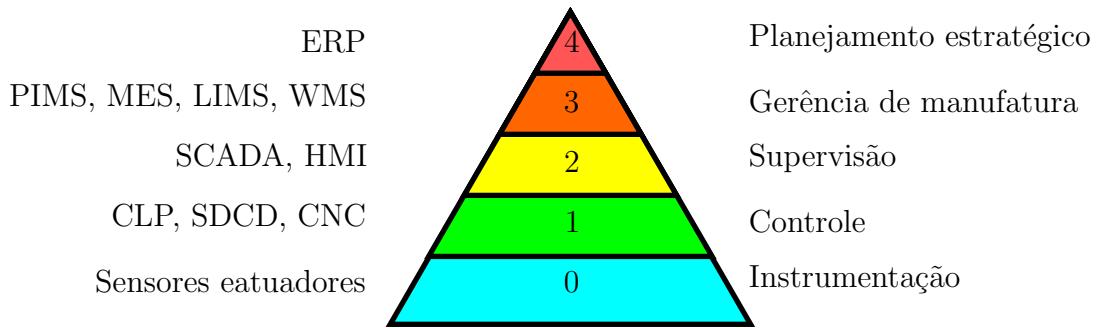


Figura 1.2: Pirâmide de automação.

Note que esta não é a única representação da pirâmide: umas começam pelo 1, outras tem apenas 4 camadas, e assim por diante, logo mais importante que o número de cada camada é o que tais camadas significam.

Nível 0 :Instrumentação Camada onde se encontram instrumentos, sensores, motores, máquinas, etc. Consistem nos equipamentos do chamado “*chão de fábrica*”.

Nível 1: Controladores Controle automático da planta – onde se localizam os Controladores Lógico-Programáveis (CLP), os Sistemas Digitais de Controle Distribuído (SDCD), os Controles Numéricos Computadorizados (CNC) e/ou computadores de controle.

Nível 2: Supervisão Supervisão e controle do processo através de Interfaces Homem-Máquina (IHMs) ou SCADA (*Supervisory Control And Data Acquisition*).

Nível 3: Gerenciamento da Manufatura Gestão dos recursos da planta e controle da produção. Sistemas PIMS (*Process Information Management System*) e MES (*Manufacturing Execution Systems*).

Nível 4: Gerenciamento da Empresa Gestão dos recursos e produção da empresa como um todo. ERP – *Enterprise Resources Planning*.

Este texto faz um estudo da automação industrial de modo *top-down*: começando do nível 3 até o nível 0. O nível 4 é mais importante para um estudo de engenharia de processo e portanto não será abordado.

Capítulo 2

Gerenciamento da manufatura: PIMS e MES

PIMS – *Process Information Management System* e MES – *Manufacturing Execution System* são sistemas da camada 3 da pirâmide de automação, responsáveis pelo armazenamento e tratamento de dados do nível 2, concentrando os dados de diversos processos separados em um único ponto. São os chamados *middleware*, pois ficam a meio caminho entre os sistemas de gerenciamento de empresa e os supervisórios, às vezes combinando funções de um ou de outro.

De forma geral, no terceiro nível da pirâmide a preocupação é em consolidar os dados brutos do processo (*data*), para com eles gerar informações (*information*) e conhecimento (*knowledge*) sobre o processo, aumentando o valor destes valores, como mostra a figura 2.1. Os dados são obtidos ou do controlador ou do supervisório de um determinado processo. A relação entre dados ou a variação destes dados no tempo geram informação sobre a planta. A relação entre informações ou a variação de informações no tempo geram conhecimento.

Um exemplo desta relação é mostrado na figura 2.2. Nesta figura, a partir dos dados de temperatura e vazão de um fluido é gerada a informação do calor removido em determinado trocador de calor. A comparação dos calores removidos de diversos trocadores gera o conhecimento de qual trocador é mais eficiente.

2.1 PIMS

Para a finalidade de gerar informação e conhecimento, o ponto de partida é obter os dados brutos. Esta é a tarefa principal do PIMS: adquirir, armazenar e apresentar diversos dados de uma planta. O PIMS foi criado e ainda é principalmente usado para processos contínuos, tais como uma refinaria ou siderúrgica, e portanto tem um enfoque muito grande em variáveis analógicas e na relação delas com o tempo.

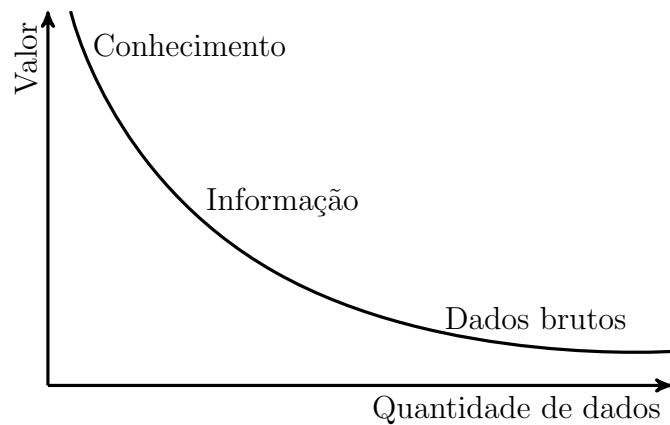


Figura 2.1: Relação entre dados, informações e conhecimentos.

Do ponto de vista do PIMS podemos esquematizar os sistemas de uma fábrica como mostra a figura 2.3, onde se vê que o PIMS tem 4 partes principais: historiador de processos, banco de dados temporal, interface gráfica e aplicações clientes (variadas funções, desde análise dos dados a interface com outros sistemas).

2.1.1 Historiador de Processos

O historiador do processo se comunica com vários sistemas do nível 1 (CLP, CNC) ou 2 (supervisório) ou ainda de outros sistemas nível 3, tais como um LIMS – *Laboratory Information Management System* ou MES para obter dados brutos dos diversos processos e acumula-os no banco de dados. Tal sistema fornece as seguintes funcionalidades:

Registro histórico para análise de incidentes, controle de qualidade, métricas de performance, entre outros.

Adequação a normas como por exemplo para controle ambiental.

Monitoração de equipamentos para controle de vida útil e apoio à manutenção.

Análise de processo facilita a visualização de dados e detecção de correlações.

Os dados coletados são principalmente os valores das variáveis do processo, sejam discretos ou contínuos, mas também abarcam outras informações, tais como a ocorrência de alarmes, a marcação de que operador está presente, o período que um equipamento está ligado, entre outros. Cada uma destas informações é identificada por um marcador único - a chamada *tag*, ao qual também está associado o

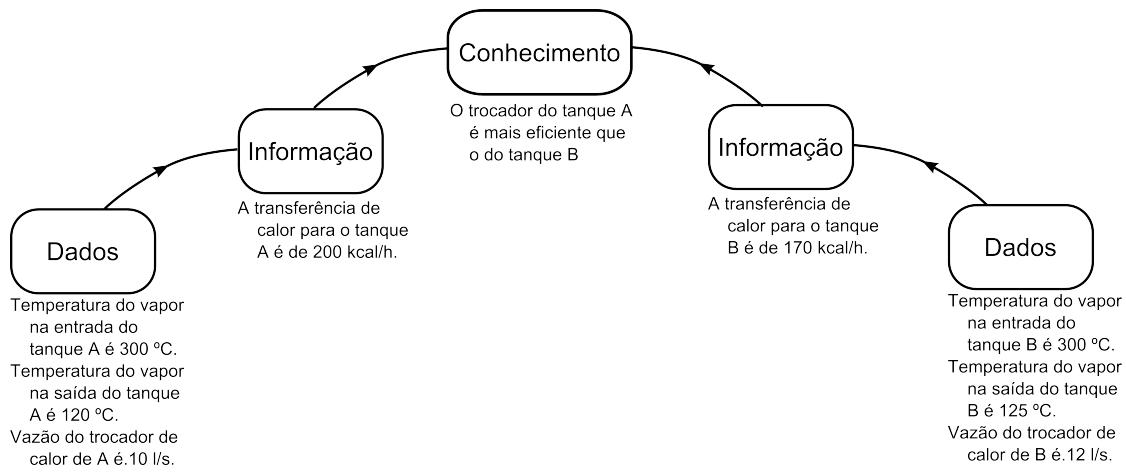


Figura 2.2: Exemplo da transformação de dados para informação e de informação para conhecimento.

endereço lógico de onde se obtém tal informação e o tempo em que tal dado foi gerado (*time stamp*). Em alguns casos se associa também uma métrica da qualidade do dado, referente a confiabilidade daquele dado, tal como se o instrumento de medida está calibrado ou não.

Estas informações podem ser obtidas tanto de sistemas SCADA (nível 2) ou de CLPs (nível 1). Algumas vantagens de pegar informação dos sistemas nível 2 são que:

- o SCADA já converteu os dados para unidades de engenharia enquanto que em alguns CLPs os dados estão em valor bruto (de 0 a 4095);
- muitas variáveis são definidas apenas no sistema SCADA, não existindo nos CLPs, tais como o motivo de alarmes ou qual operador está monitorando a operação;
- interface com os sistemas SCADA costuma ser padrão, o que facilita a comunicação.

Vantagens de obter as informações do CLP são:

- busca dos eventos com menor atraso temporal;
- pode-se coletar os dados em um ponto único, se todas as redes de CLPs estiverem interligadas;
- CLPs são mais confiáveis e apresentam menor suscetibilidade a falhas que os sistemas SCADA.



Figura 2.3: Sistema PIMS.

2.1.2 Banco de Dados Temporal

A maioria das análises realizadas nos dados de um sistema PIMS são em função do tempo, logo é comum ele usar um banco de dados que indexa a informação pelo tempo. Basicamente é uma tabela, relacionando *time stamp*, *tag*, tipo de dado (análogo, booleano, texto), valor e qualidade (se houver).

Um problema do uso deste tipo de banco de dados, ao invés dos chamados bancos de dados relacionais, tipo SQL, é que a busca por informação pode ter uma baixa performance quando a quantidade de dados aumenta muito. Esta é a principal razão para que estes sistemas façam uma compressão de dados, que basicamente se resume a não armazenar dados que não tragam muita informação nova. A figura 2.4 mostra a idéia por trás da compressão de dados.

Algoritmos comuns para a compressão de dados no PIMS são *banda morta*, onde os dados são apenas armazenados se variarem mais do que um mínimo especificado; o *SDCA – Swinging Doors Compression Algorithm*, onde para cada valor recebido é definida uma reta entre ele e o último valor armazenado, descartando valores que possam ser definidos por esta reta e mais um erro; e o *boxcar/backslope*, que usa a banda morta e mais uma reta definida pelo último valor armazenado.

2.1.3 Interface Gráfica

A interface gráfica de um sistema PIMS é, em muitos aspectos, muito parecida com a de um sistema SCADA, contendo representações pictóricas do processo (sinóticos) com os valores de várias variáveis e gráficos de tendência. Tais elementos são melhor vistos no contexto de um sistema SCADA.

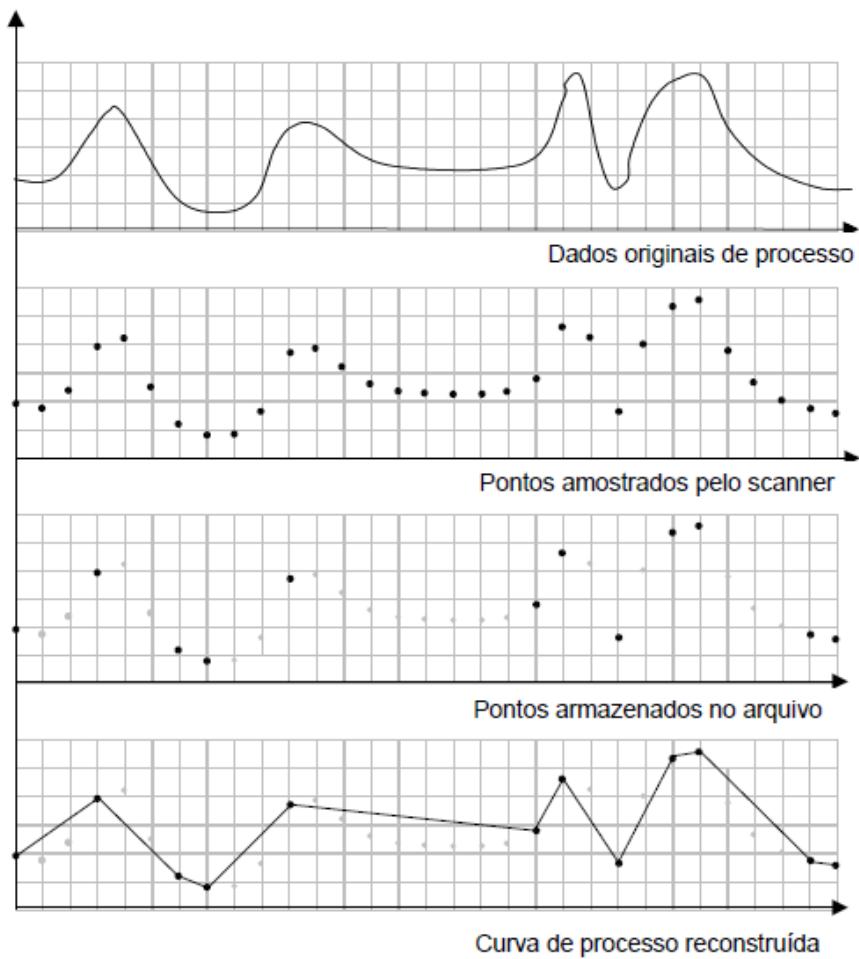


Figura 2.4: Processo de amostragem, compressão e reconstrução dos dados.

2.2 MES

Assim como o PIMS, um sistema MES também adquire dados do processo com o objetivo de apresentar uma visão geral do processo. Porém, enquanto o PIMS é focado mais no armazenamento dos dados, o MES tem uma gama maior de funções e um papel mais ativo. Historicamente, os MES são usados principalmente em processos discretos, muitas vezes em sistemas de automação flexível ou programada. Isto faz com que o MES tenha que lidar com o sequenciamento dos processos, o que ocorre menos em sistemas contínuos.

Os sistemas MES agregam diversas funções de sistemas anteriores mais simples e específicos e são definidos por terem as seguintes funções:

Gerenciamento das definições de produto. Todas informações necessárias para a fabricação do produto. Isto inclue lista de materiais e insumos, set-points do processo e receitas.

Por receita, entenda-se uma variação do processo para, na mesma máquina, produzir produtos diferentes ou variações dele. Por exemplo, num processo de pintura a receita diria quais as tintas usadas, em que sequência, em que volume e em que velocidade de aplicação. É muito importante na automação flexível e programável.

Sistemas de automação fixa também se utilizam de receitas. Podemos citar dois casos mais importantes: para a melhoria do processo e do produto, testando diferentes receitas e checando os resultados e para casos de variações de matéria prima ou de condições ambientais, onde se buscaria a melhor receita para cada caso.

Gerenciamento de insumos. Permite preparar e executar ordens de produção com garantia de disponibilidade dos insumos. Esta função cuida do controle de estoques e dos pedidos aos fornecedores, principalmente quando se usa a metodologia de *just in time*, que minimiza os estoques.

Agendamento de produção. Permite determinar a ordem que a produção será feita, para alcançar os requerimentos de produção definidos pela ERP (camada 4 da pirâmide) utilizando otimamente os recursos. Também chamado de *scheduler*.

A figura 2.5 mostra o exemplo de um *scheduler*, onde se vê o uso de um diagrama de Gantt para definir o sequenciamento das operações e máquinas.

Tipicamente inclui ferramentas de simulação, que permitem comparar diversas opções de ordens e estimar efeitos quando de mudanças imprevistas na sequência de produção (em geral por conta de uma parada não programada).

Envio de ordens de produção. Em função do agendamento feito, o MES cuida de enviar as ordens de produção para os diversos postos da planta. A informação vai para os supervisórios e, em alguns casos mais avançados, para os controladores.

Acompanhamento da execução de ordens de produção. Também se comunica com sistemas níveis 1 e 2 para garantir a execução das ordens. Inclui também o registro de paradas.

O registro de paradas é feito automaticamente quando o equipamento para, seja por alguma condição espúria detectada no nível 1 ou por ação do operador no nível 2. Em ambos os casos fica registrado uma parada em aberto, que

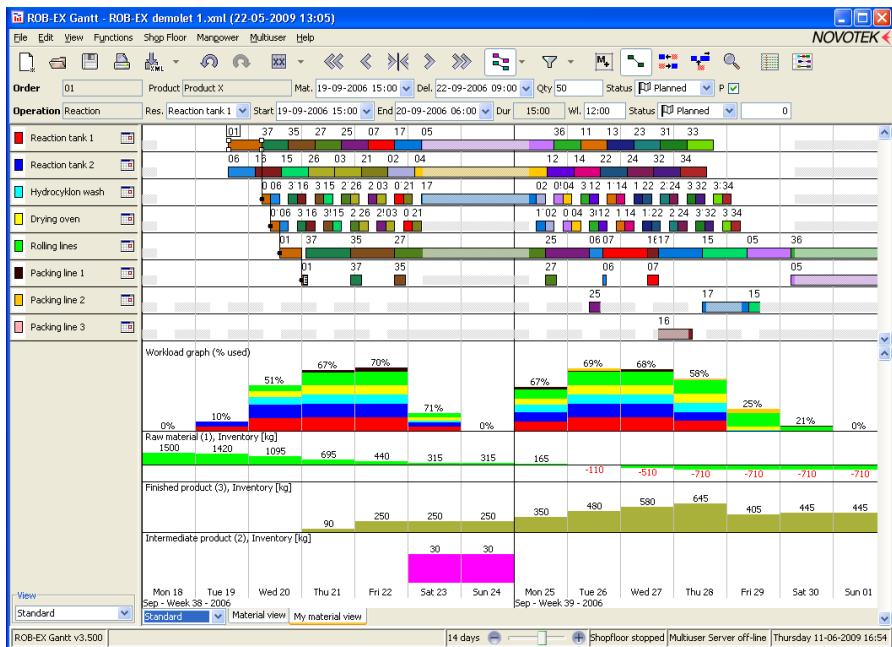


Figura 2.5: Exemplo de um *scheduler*.

só finaliza quando o operador complementar certas informações que auxiliam no diagnóstico da parada e o equipamento voltar a funcionar.

Coleção dos dados de produção. Equivalente ao historiador de processo do PIMS.

Análise da performance da produção. Cálculo dos chamados índices de produção – *KPI*, *Key Performance Indicators*, tais quais na figura ???. É a geração de informação a partir dos dados da produção.

O OEE – *Overall Equipment Effectiveness* é um exemplo de KPI não diretamente ligado ao produto, mas à produção. O OEE aponta a efetividade de um determinado equipamento ou célula de produção. Este índice é dado por:

$$\text{OEE} = \text{disponibilidade} \times \text{performance} \times \text{qualidade}, \quad (2.1)$$

onde por disponibilidade entende-se a razão entre o quanto de tempo o equipamento funcionou e o quanto de tempo ele deveria ter funcionado, ou seja, descontando-se as paradas não programadas; performance é a razão entre a produção do equipamento enquanto funcionava e a capacidade de produção

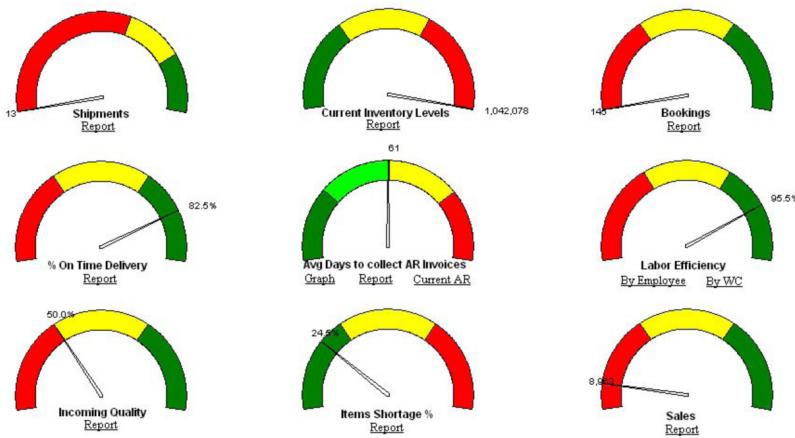


Figura 2.6: Exemplo de visualização de KPIs.

de que o equipamento é capaz, ou seja, descontando a ociosidade do equipamento; e qualidade é o valor do que o equipamento produziu em relação ao valor se não tivesse havido nenhum descarte ou geração de produtos de menor valor. Esta relação é melhor visualizada na figura 2.7 e exemplificada na figura 2.8.

■ **OEE = Disponibilidade * Performance * Qualidade**

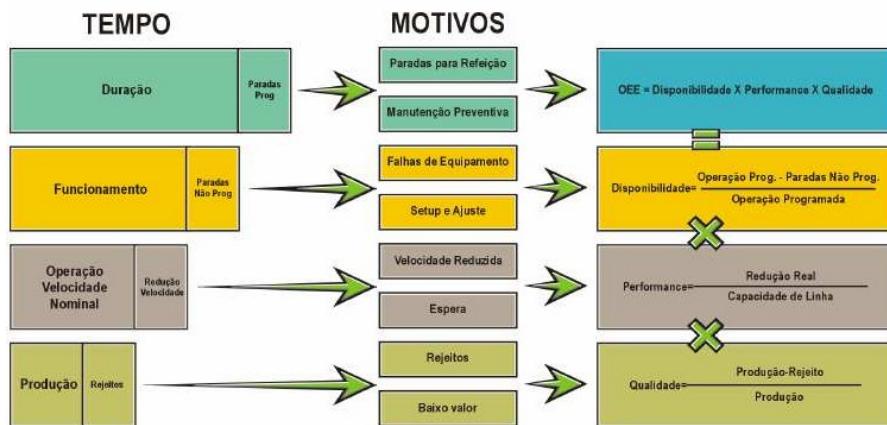


Figura 2.7: Overall Equipment Effectiveness.

Rastreamento da produção. Permite levantar que produto ou lote foi feito quando e em qual equipamento. Útil para melhoria da produção e imprescindível para remédios e produtos alimentícios.



Figura 2.8: Overall Equipment Effectiveness.

Armazenamento dos logs de produção. Hoje em dia tais logs são inseridos pelo operador no próprio sistema supervisório e realcionados às variáveis de produção.

Interface de auditoria. Permite a análise dos diversos dados e informações armazenados e o cruzamento destes dados com outras bases de dados.

2.2.1 Redes industriais

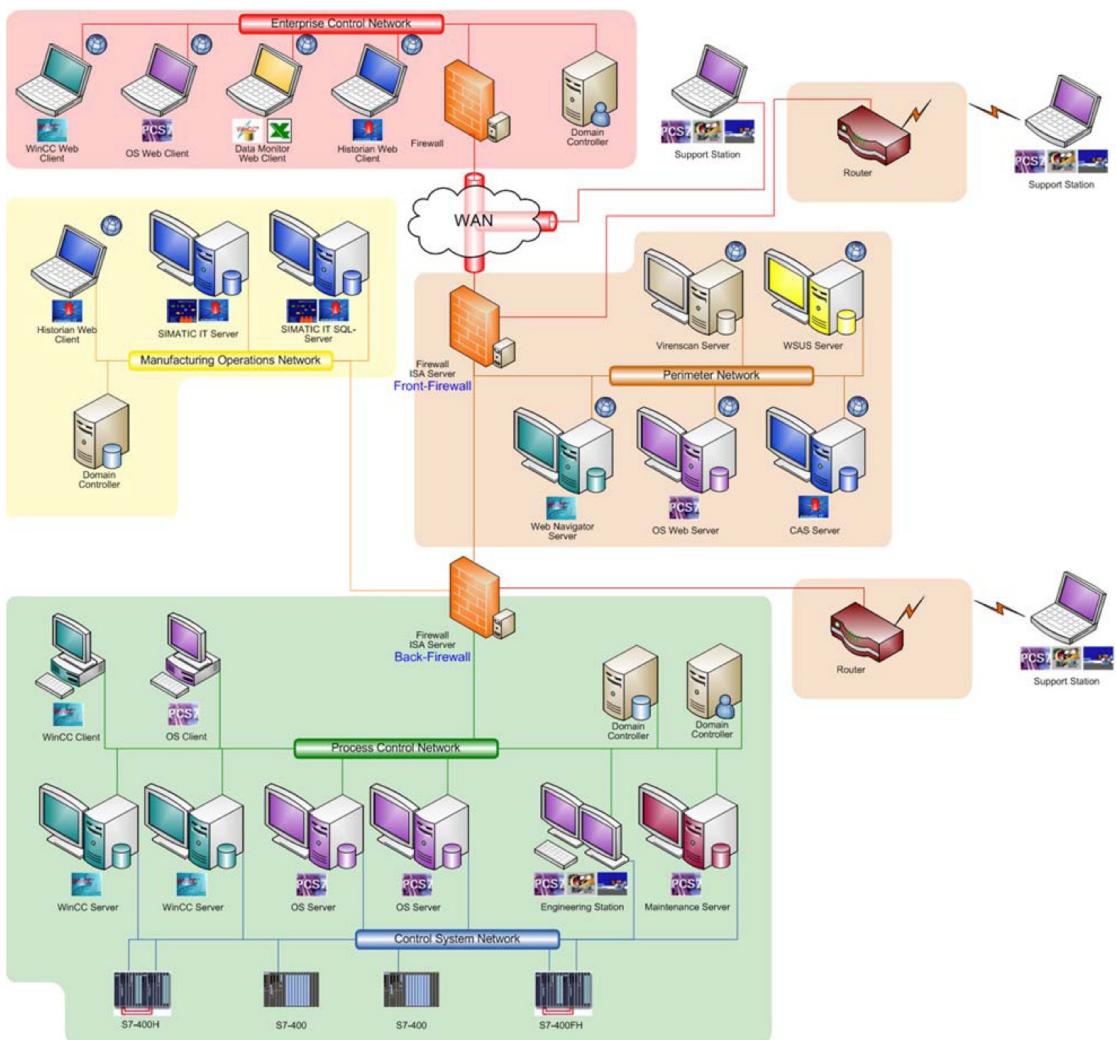


Figura 2.9: Arquitetura de rede de dados industrial.

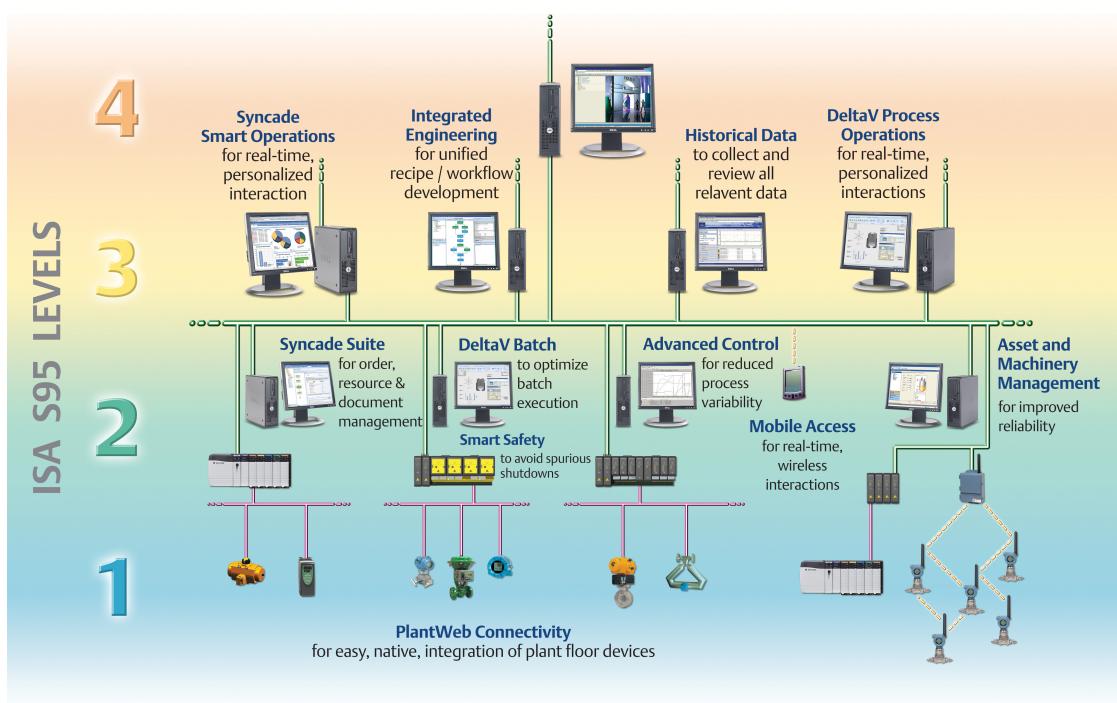


Figura 2.10: Arquitetura de rede de dados industrial.

2.3 Gerenciamento da manufatura: MES

2.4 Redes de Comunicação: Introdução e noções básicas

Na pirâmide de automação, a partir da camada 2, os sistemas utilizados são softwares em computadores, sejam servidores, terminais, desktops, notebooks ou mesmo tablets e smartphones (para o caso do supervisório). No nível de controle, o dispositivo utilizado pode ser um computador, um CLP – que não deixa de ser um computador – ou outro dispositivo que incorpora um processador. Hoje em dia até no nível 0 estão ficando cada vez mais comuns os dispositivos ditos *inteligentes*, que incorporam também um microprocessador. A comunicação entre eles é feita por uma rede de dados, tais como a ethernet.

O problema da comunicação de dados automática entre diversos dispositivos é bastante complexo. Ele envolve a:

1. transferência de dados de um dispositivo a outro,
2. em alta velocidade,
3. sem perda de informação,
4. eventualmente passando por dispositivos intermediários,
5. talvez envolvendo sistemas de comunicação diferentes,
6. com privacidade,
7. sem envolver outros dispositivos desnecessariamente.

Peguemos como exemplo pagar uma conta via internet. Logo é necessária a comunicação do dispositivo caseiro (que seja um tablet) e o computador central do banco (1). Quanto mais rápida for esta comunicação, mais agradável é para o usuário e mais barato é para o banco (2). Obviamente nenhuma das partes envolvidas querem que seja pago o valor errado ou a conta errada (3). Entre o tablet e o servidor do banco estão: o modem wifi, a central telefônica, servidores da companhia telefônica e talvez de outras fornecedoras de serviço (4), e em geral não é do interesse do usuário que a companhia telefônica ou outro elemento saiba de sua senha do banco (6). A comunicação do tablet ao modem é por wifi. Do modem para a central telefônica é pelo fio telefônico. Da central em diante pode ser por cabos de cobre ou, mais provavelmente, por fibra óptica (5). Também, por questão de custos e privacidade, não é desejável que outros elementos recebam aquela informação, mesmo que criptografada.

Para resolver este problema, foi desenvolvida a idéia de quebrá-lo em várias partes, ou camadas, onde cada camada é responsável por uma determinada parte

do problema. A informação fica então passando de camada a camada para resolver todas as questões relativas à comunicação. Tal esquema é também chamado de pilha de protocolos.

2.4.1 Modelo OSI

Diferentes tipos de rede de dados tem problemas diferentes. O modelo OSI procura definir camadas o mais genéricas possíveis, de modo a abranger qualquer tipo de comunicação de dados. Este modelo define 7 camadas: física, enlace, rede, transporte, sessão, apresentação e aplicação.

Física

A camada física é a única que não pode ser implementada em software. Ela lida com as definições eletro-mecânicas necessárias para a comunicação. Ou seja, a camada física define:

- a forma como os dados são transmitidos (sinais elétricos por cabo, sinais de rádio, luz, etc);
- as características do meio de transmissão, tais como o tipo de cabo, os níveis de tensão, o formato dos conectores, etc;
- a taxa de transmissão.

Note-se que a taxa de transmissão acaba sendo limitada justamente pelos diversos parâmetros físicos descritos pela camada física de um determinado protocolo de comunicação.

Outro parâmetro definido na camada física é se a rede é *simplex*, *half-duplex* ou *full-duplex*. Simplex significa que a comunicação no meio de transmissão é apenas num sentido (exemplo: televisão), half-duplex é uma comunicação em dois sentidos, mas apenas um por vez (exemplo: walkie-talkie) e full-duplex é uma comunicação em ambos os sentidos ao mesmo tempo (exemplo: telefonia).

Enlace

A camada de enlace cuida da comunicação de um dispositivo a seu vizinho, que não necessariamente são os pontos finais.

No exemplo acima, esta seria a comunicação entre o tablet e o modem wifi; entre o modem e a central; entre a central e o servidor da companhia telefônica, e assim por diante.

A camada de enlace de um protocolo define quem acessa o meio de transmissão e quando, que sinais indicam o início e o fim de uma transmissão, mecanismos que

detectem e/ou corrijam erros e, se necessário, para qual dentre vários dispositivos é aquela comunicação específica.

Alguns protocolos de comunicação, tais como USB e ethernet, fazem a conexão ponto a ponto. Neste caso apenas 2 dispositivos podem acessar o meio. Outros sistemas, como o wifi, podem ter vários (em alguns casos, centenas) de dispositivos no mesmo meio. O que torna necessário a definição de quem pode acessar o meio em cada instante. Várias possibilidades existem:

mestre-escravo Neste sistema, a comunicação sempre é iniciada pelo mestre, e um escravo só ocupa o meio quando o mestre requisita uma informação. Ex.: USB, Modbus.

token ring É passada uma ficha (token) virtual entre os dispositivos. Quem tem a ficha pode falar.

Multiplexação por tempo Cada dispositivo tem uma hora específica para controlar o barramento. Um sinal periódico (*NUT- Network Update Time*) sincroniza os dispositivos.

CSMA-CD – *Carrier Sense Multiple Access with Collision Detection*
Sempre que um dispositivo quer se comunicar com outro, ele checa antes se o meio está livre. Ex.: wi-fi, bluetooth, CAN.

Tipicamente os protocolos da camada de enlace acrescentam uma certa redundância ao dado transmitido, o que permite detectar e até corrigir erros de transmissão. São os chamados códigos corretores de erro, dentre os quais se utiliza bastante o CRC - *Cyclic Redundancy Check*.

Rede

Camada responsável pelo roteamento da informação. Ou seja: Se A quer se comunicar com D, mas A apenas tem ligação direta com B e C, para quem A deve mandar a informação?

Tipicamente este roteamento é feito através de uma tabela que mapeia o endereço lógico de um dispositivo (o endereço usado nas etapas acima, tais como o IP) e o endereço físico, usado pela camada de enlace (muitos sistemas usam o *MAC address*). Esta é a última camada a que uma comunicação chega num dispositivo que não o inicial ou final: o programa desta camada checa se o endereço lógico da mensagem é o mesmo daquele dispositivo; não sendo, ele busca na tabela qual o endereço físico para o qual reenviar aquela mensagem e reenvia, tal qual mostra a figura 2.11. Isto serve também para fazer a conexão entre diferentes redes.

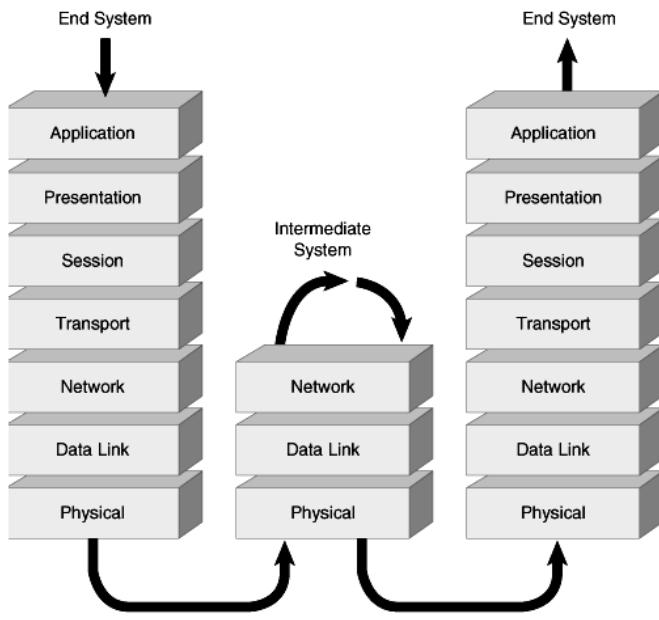


Figura 2.11: Comunicação passando por dispositivo intermediário.

Transporte

A camada de transporte cuida da comunicação entre o ponto inicial e final. Usa apenas o endereço lógico. Esta camada pode resolver quebrar uma mensagem grande em vários pacotes, mandando um pacote de cada vez para a camada de rede e remontando a mensagem no recebimento.

Um conceito interessante utilizado na camada de transporte é o de *Quality of Service – QoS*, qualidade de serviço. Um sistema com alto QoS garante a chegada de todos os pacotes, na ordem em que foram enviados.

A internet foi originalmente pensada como um sistema de baixo QoS, voltado para a transmissão de arquivos. A ordem dos pacotes não interessa e nem o atraso de um pacote. Caso um pacote se perca, simplesmente pede-se que seja reenviado. Uma rede de telefonia, por outro lado, busca garantir que o tempo de cada pacote seja o mesmo, para não piorar a qualidade da voz transmitida.

Para diversas aplicações industriais, o QoS é importante: não se pode aceitar que uma mensagem de alarme, por exemplo, demore muito a chegar. EtherCAT - *Ethernet for Control Automation Technology* é uma modificação de Ethernet para mensagens com diferentes QoS.

Sessão

Esta camada cria, gerencia e termina conexões entre 2 pontos de uma rede. É mais comum na telefonia, onde se gera um caminho fixo para uma determinada ligação. Basicamente esta camada gera a tabela que é usada pela camada de rede.

Apresentação

Faz a mudança no formato dos dados. Exemplo: troca de quebra de linha entre sistemas Unix e Windows, mudança de codificação windows 1252 para UTF, etc.

Inclui também a criptografia, que é a base para garantir a privacidade da comunicação ponto-a-ponto.

Aplicação

A aplicação final

2.4.2 Internet

A Internet define uma pilha de apenas 4 protocolos: o de enlace, que engloba também o físico; o IP – *Internet Protocol*, equivalente ao de rede do modelo OSI; o de transporte, que pode ser um entre vários, sendo o TCP e o UDP os mais comuns e o de aplicação, que envolve sessão, apresentação e aplicação num só.

A figura 2.12 mostra a sequência que é feita com uma informação passada por http pela internet.

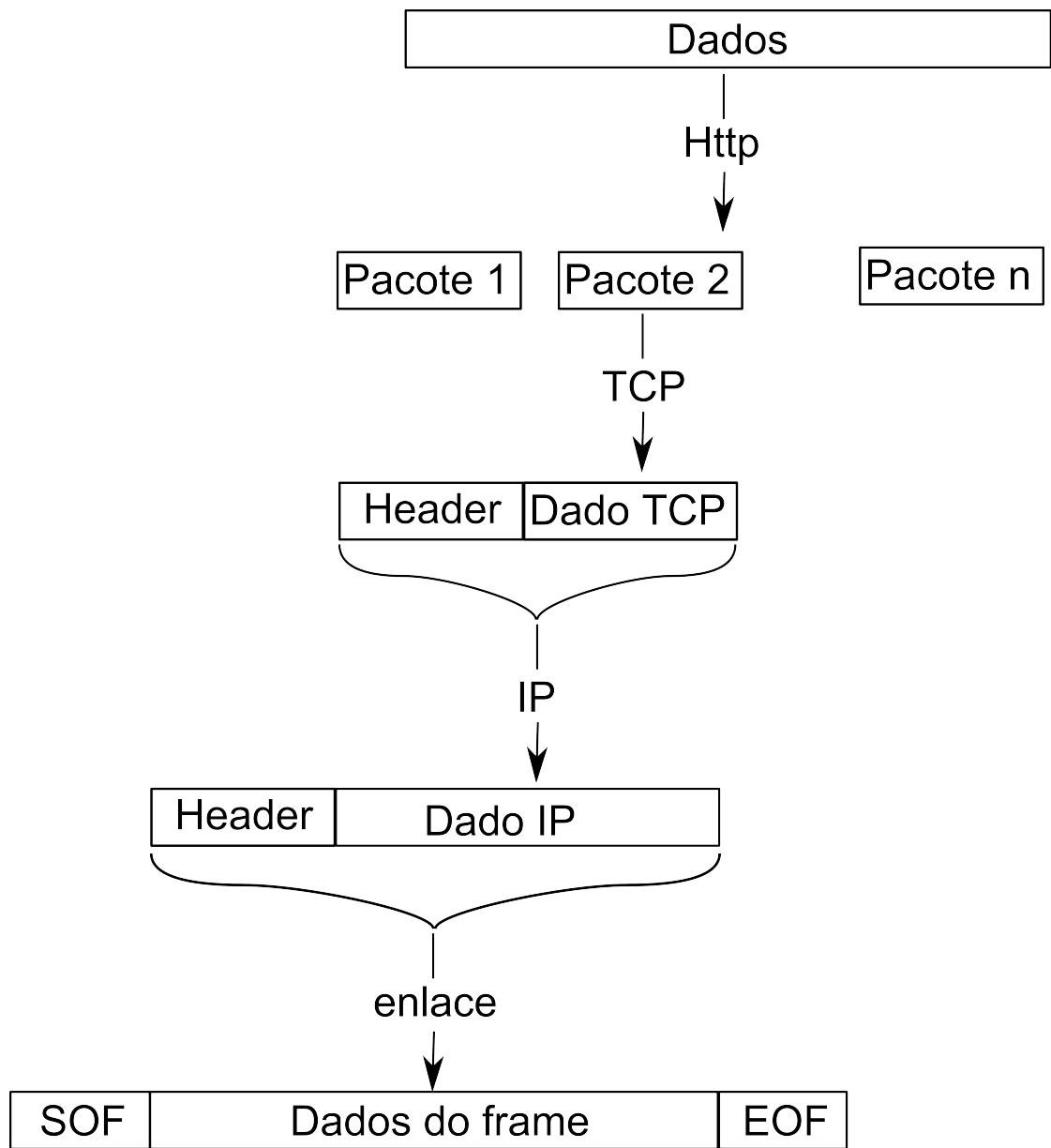


Figura 2.12: Encapsulamento da informação na pilha TCP/IP

Capítulo 3

SCADA – Supervisory Control and Data Acquisition

- 3.1 Arquitetura de sistemas SCADA e interfaceamento com níveis de automação**
- 3.2 Funcionalidades principais de sistemas SCADA**

Capítulo 4

Sistema SCADA MANGO

Mango é uma solução SCADA/HMI open source que roda em um sevidor internet java. Logo toda interface dele é através de um navegador. Muito embora ele seja open source, existem versões aprimoradas dele que apresentam umas facilidades a mais, mas a versão básica já permite fazer um sistema SCADA completo.

4.1 Instalação

Vide <http://infiniteautomation.com/wiki/wiki.php>, a instalação é simplesmente descompactar um arquivo zip e rodar um script para abrir o programa. Basta ter o java 1.7 instalado.

4.2 Uso

Ao rodar o mago, ele abre a tela mostrada na figura 4.1. A página do mango pode ser acessada novamente pelo endereço <http://localhost:8080>. O login padrão é **admin**, senha **admin**.

O topo da tela do mango pode ser visto na figura 4.2. Na parte mais superior tem os indicadores de eventos e alarmes, seguido dos links para as diversas áreas do mango.

Um dos primeiros pontos a ser definido é o das fontes de dados do sistema. Chega-se nisto clicando no ícone *data sources*, vide figura 4.3. São possíveis diversos tipos diferentes de fontes de dados, mas para nós no momento os que vão interessar são o **Modbus I/P**, o **Modbus Serial** e o **Virtual Data Source**, este último principalmente para simulação.

Para cada fonte de dados, são criados pontos (*data points*), correspondentes a uma tag. No caso de uma fonte virtual, há opções quanto ao tipo de dado (binário, multi-estado, numérico, texto) e quanto ao valor automático, vide figura 4.4.

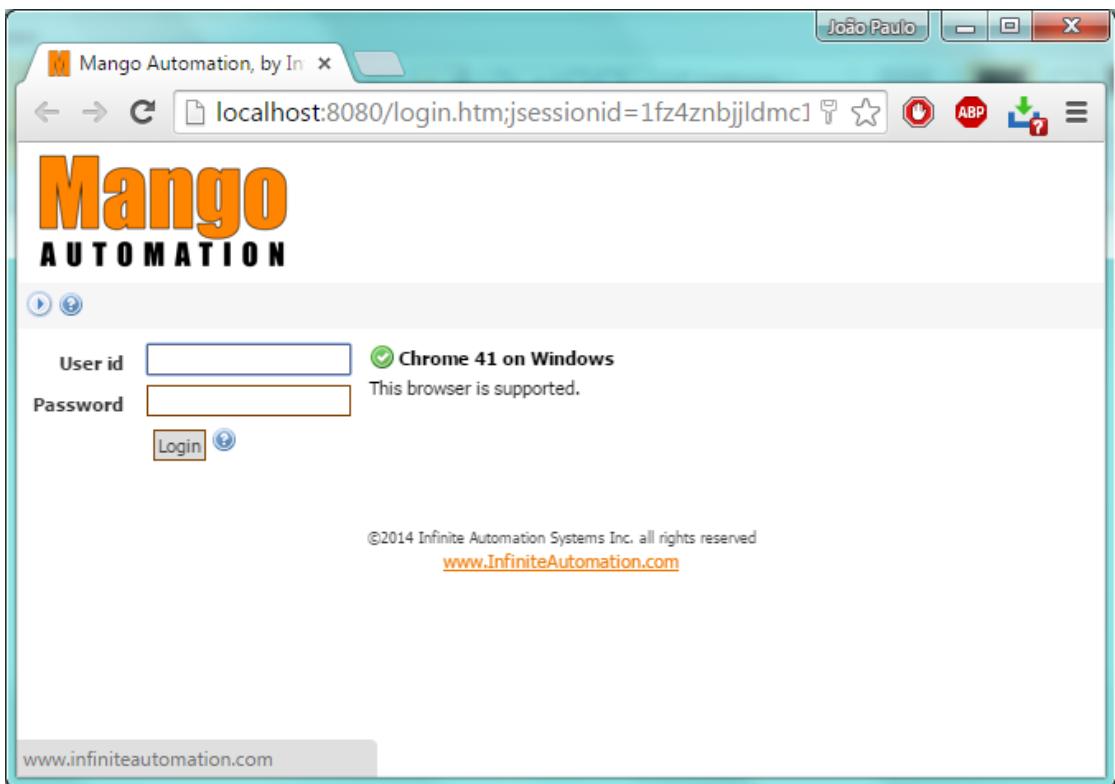


Figura 4.1: Tela de login do mango.

O protocolo modbus é um dos utilizados na automação industrial. Existe uma implementação do modbus serial para arduino, que é implementado pelo uso da biblioteca **SimpleModbusSlave**, que faz o arduino trabalhar como um escravo modbus. O data source Modbus Serial nos permite então obter dados de um arduino, seguindo este protocolo.

A configuração do Modbus para a comunicação com o arduino é tal como mostrado na figura 4.5. É importante que a porta seja a mesma do arduino, bem como bit rate (baud rate), data bits, stop bits, stop bits e parity. O encoding deve ser do tipo RTU e deve se marcar a caixa **Contiguous batches only**.

O modbus trabalha endereçando o escravo pelo *slave id*, que no caso usamos 1. Os dados são armazenados nos chamados registradores (*registers*). Enquanto o Modbus especifica 4 tipos de registradores, o arduino implementa apenas o *Holding register*, logo nossos pontos devem ser todos deste tipo e concordar com a sequencia definida no arduino. Cada registrador é de 16 bits. Podemos definir um ponto binário como sendo um único bit daquele registrador ou um ponto numérico, que usa todos os 16 bits.

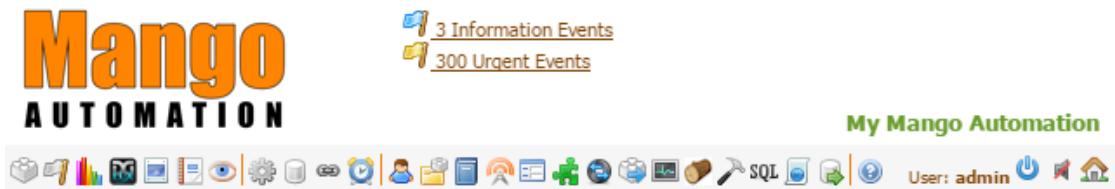


Figura 4.2: Topo da tela do mango.

4.3 *Watch list*

Podemos observar o comportamento dos pontos criados na tela *Watch lists* (figura 4.6). Se o ponto for configurável (settable), também podemos fazé-lo por esta tela.

4.4 Tela gráfica

A interface normal com o operador é feita através do sinótico – a representação gráfica do processo. O mango permite montar uma tela gráfica apresentando os diversos valores das variáveis e os controles, como mostra a figura 4.7. A apresentação de variáveis é de forma bem direta, através dos diversos comandos de gif analógico, gif binário, gif dinâmico e gráficos. O ajuste de valores, porém, é um pouco mais complexo.

Pode-se ajustar o valor de um ponto no mango marcando a opção **Exibir controles**, a partir do que se pode abrir uma caixinha que permite a escrita do valor na variável. Esta opção porém é mutio pouco prática.

Outra opção mais interessante é através dos *server side scripts*, ou scrpts do servidor. Nestes casos define-se um comando em javascript que permite ler e alterar o valor de uma variável. A página <http://infiniteautomation.com/wiki/doku.php?id=graphics> mostra vários exemplos de códigos úteis para esta tarefa.

Um adendo: os códigos da página supracitada consideram os endereços a partir da pasta `<mango>/web`. Porém o arquivo zipado do Mango não contém aí a subpasta Graphics, o que faz com que os exemplos não funcionem. é necessário copiar a pasta `<mango>/web/modules/sstGraphics/web/graphics` para este local. Pode-se (e deve-se) acrescentar outras imagens mais interessantes a este diretório.

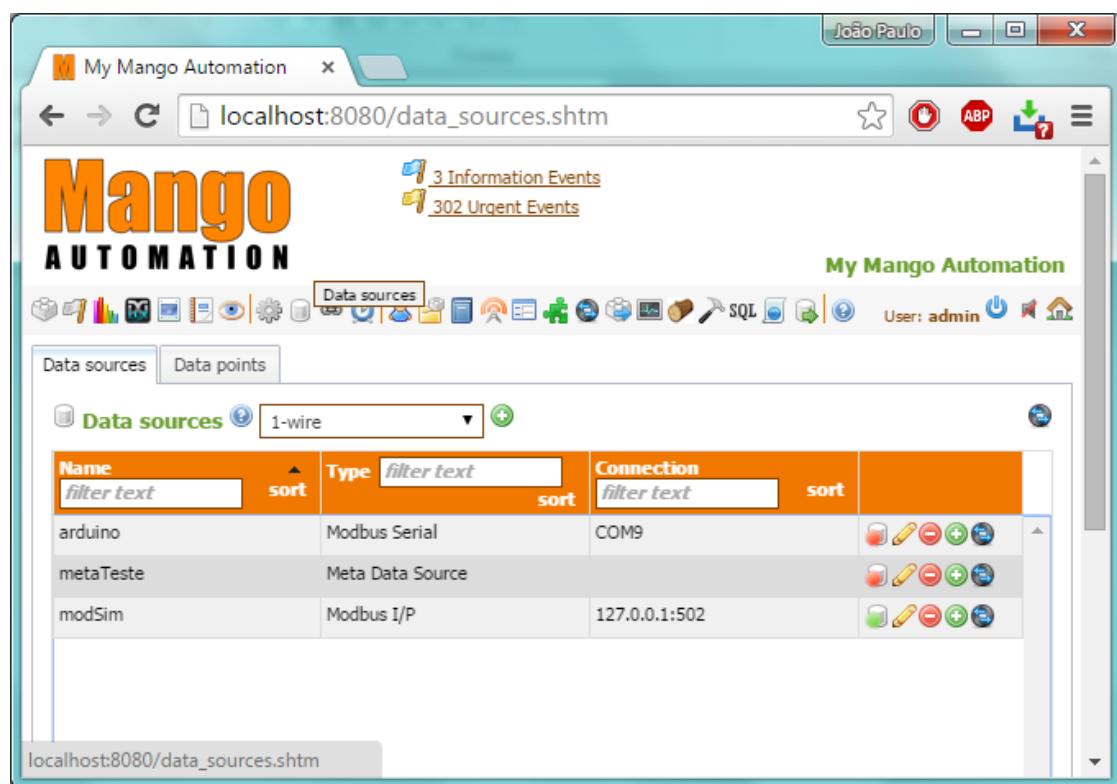


Figura 4.3: Tela de fontes de dados.

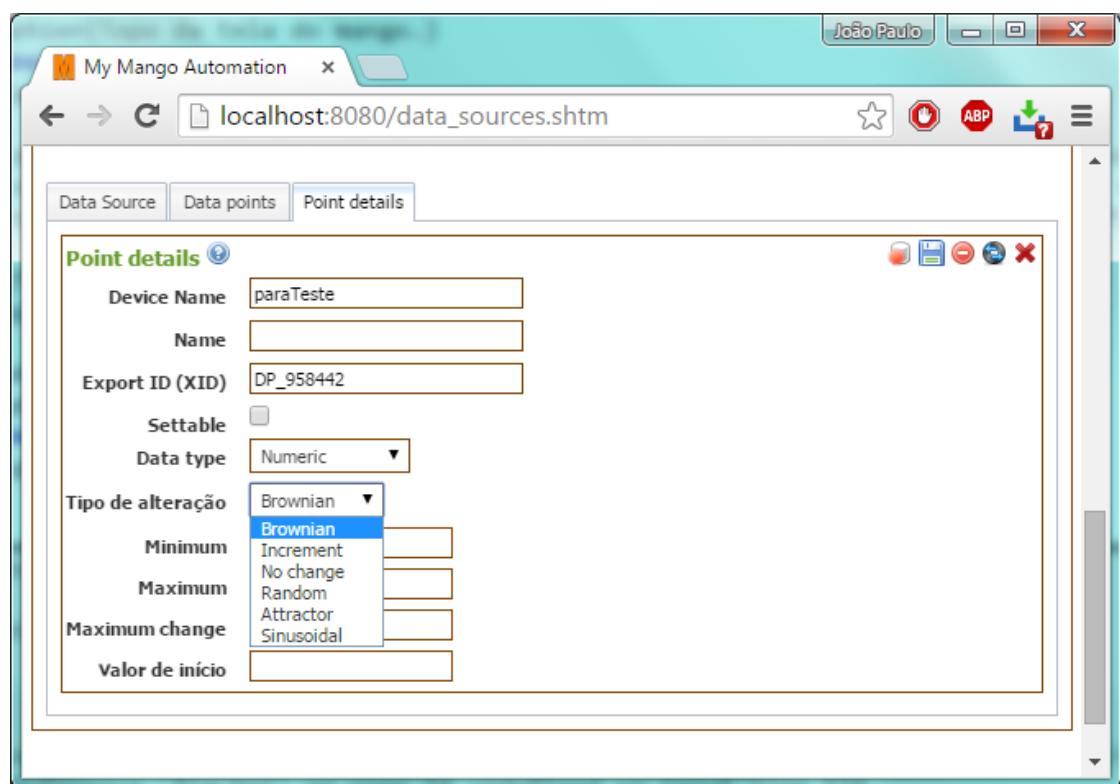


Figura 4.4: Definição de dados virtuais para simulação.

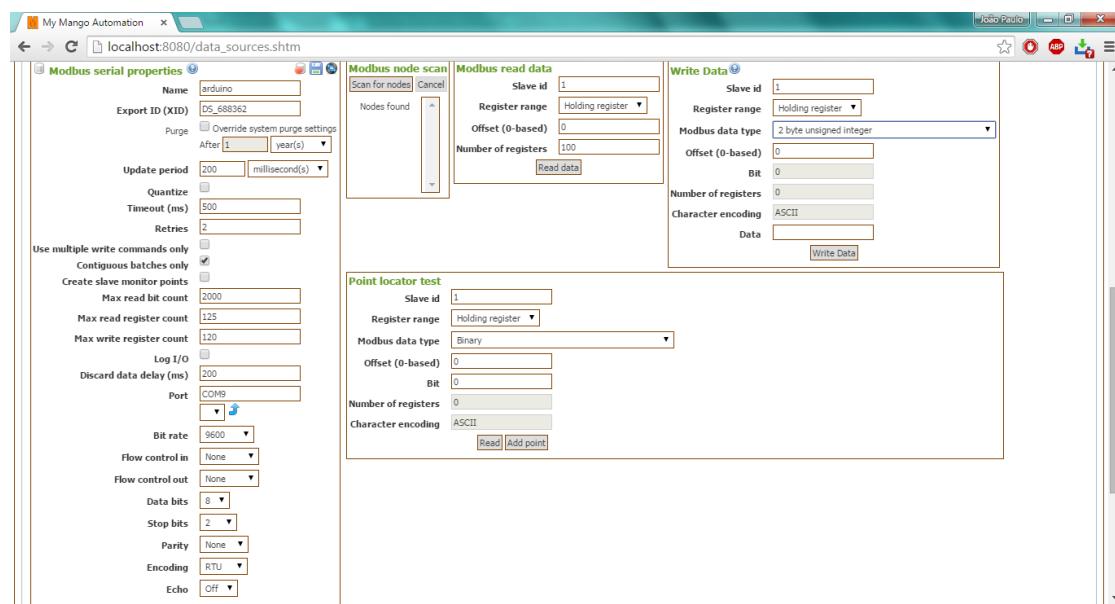


Figura 4.5: Definição de uma fonte de dados do tipo modbus serial.

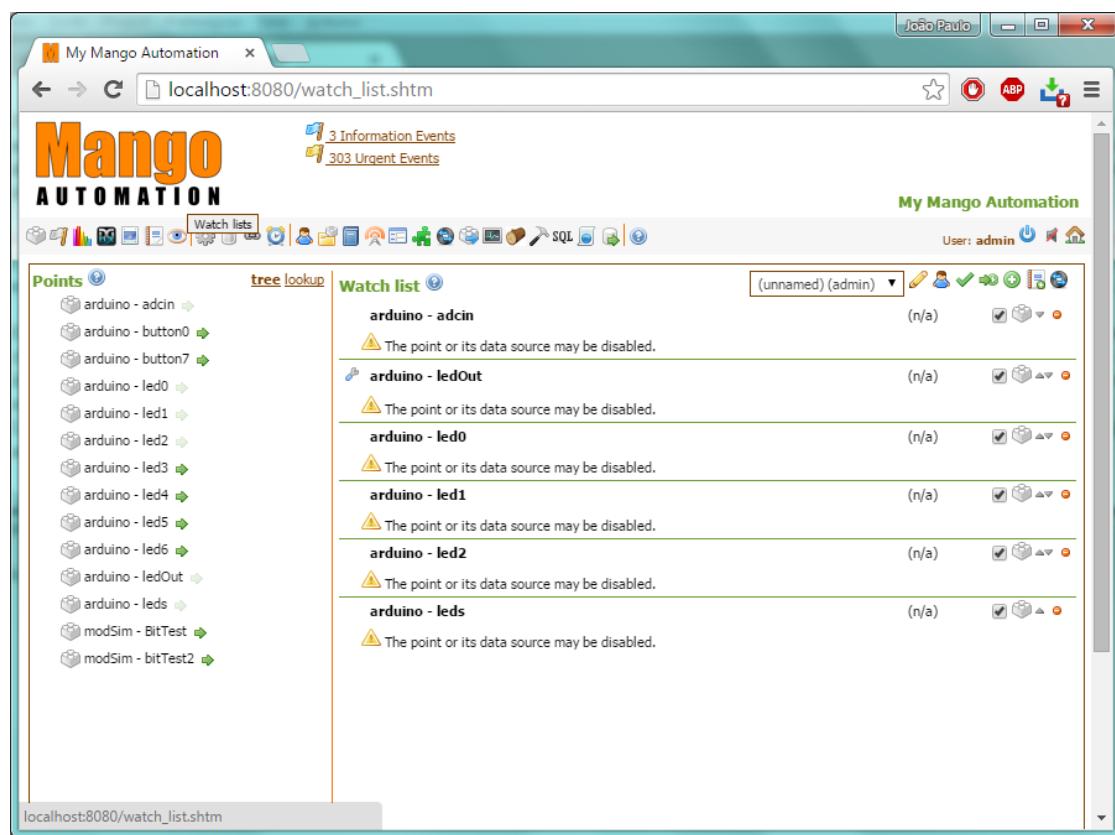


Figura 4.6: *Watch lists* para observar as variáveis.

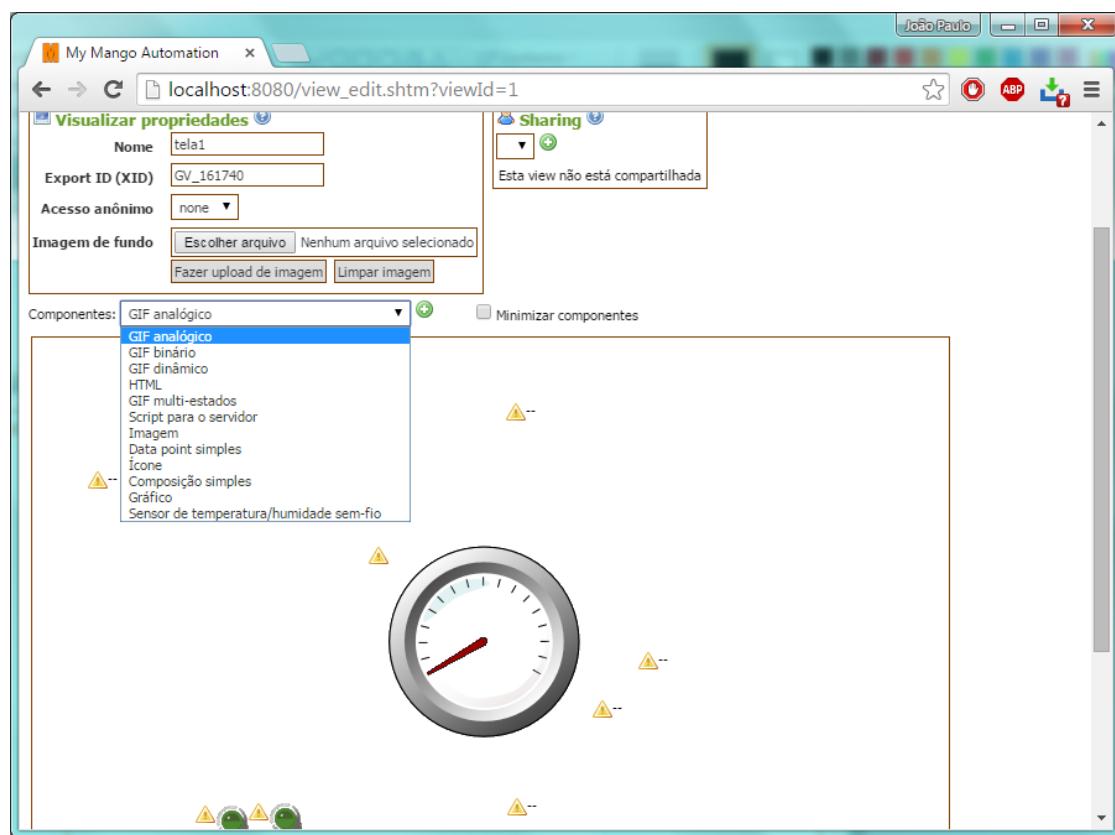


Figura 4.7: Montagem de sinótico.

Capítulo 5

Controladores Lógico-Programáveis (CLP)

Capítulo 6

Programação - arduino

O arduino é uma plataforma de microcontrolador simplificada. O nome arduino refere-se a: uma placa com um microcontrolador Atmel, uma linguagem de programação e um ambiente de desenvolvimento para esta linguagem. E também um auto-proclamado rei da Itália, mas este último não importa para nós.

A placa Arduino tem um conector USB para se ligar ao computador. Isto serve tanto para programar o microcontrolador quanto para comunicação entre os 2. Existem várias versões do arduino, pois já que é um sistema *open-source* quem quiser pode fazer sua versão diferente da placa. Vamos nos referenciar aos arduinos UNO ou outras placas compatíveis com ele.

O arduino UNO tem 4 barras de pinos fêmeas para conexão com outros dispositivos: uma com tensões de alimentação (POWER), um com 6 entradas analógicas (ANALOG IN) e 2 com um total de 14 entradas e saídas digitais (DIGITAL).

A linguagem de programação arduino é basicamente a linguagem C++ para microcontroladores ATME, mas com algumas funções e definições facilitadoras. A principal diferença entre C++ e a linguagem arduino é que não existe a função main(), mas sim as funções (ou rotinas) setup() e loop(). A função setup() é executado apenas uma vez no momento que o arduino é ligado (ou resetado) e depois o código dentro da função loop() é executado repetidamente. Com isto o esqueleto de um programa arduino fica:

Código 6.1: Esqueleto de um programa arduino.

```
void setup() {
    //codigo a ser executado no inicio
}

void loop() {
    //codigo a ser executado repetidamente
}
```

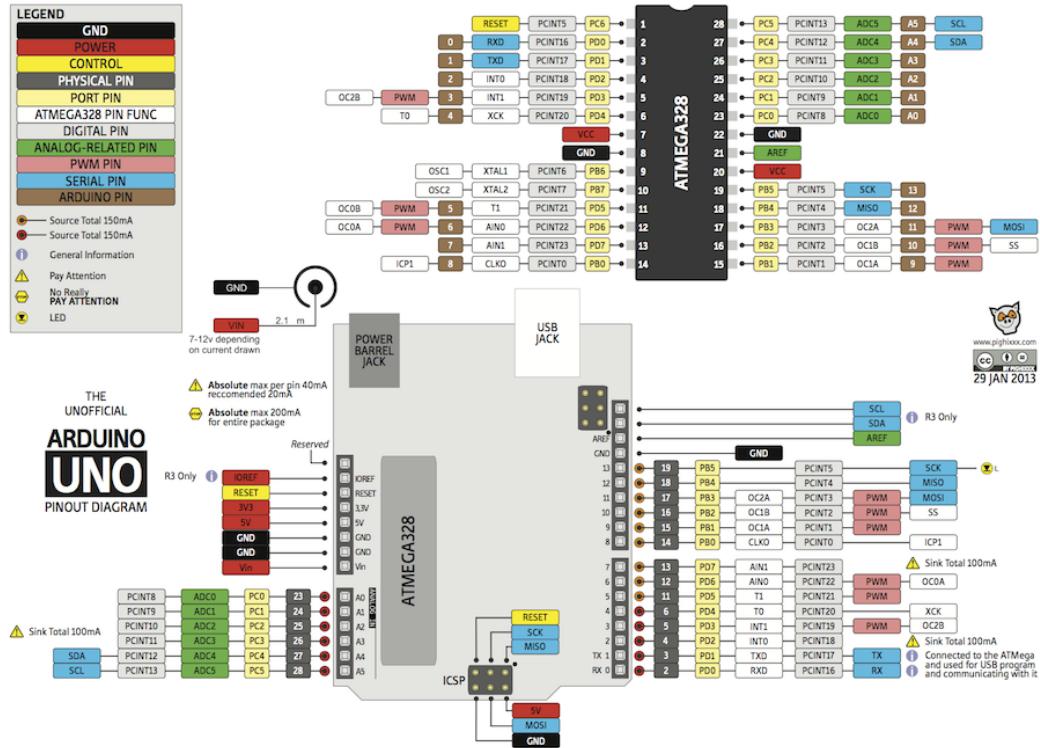


Figura 6.1: Pinagem do Arduino Uno

Lembrando que no arduino, como no C, tudo que tiver depois de // é comentário e é ignorado pelo compilador.

6.1 Piscando um led.

Vamos passar logo a um exemplo para analisar um programa arduino. As placas de arduino UNO já tem um led ligado ao pino 13, identificado por um L na placa. Podemos fazer um programa que faça este led piscar.

Código 6.2: Programa para piscar led.

```

void setup() {
    //codigo a ser executado no inicio
    pinMode(13,OUTPUT); //define o pino 13 como uma saida (led)
}

void loop() {
    //codigo a ser executado repetidamente
}

```

```

digitalWrite(13,LOW); //apaga o led
delay(500); //espera meio segundo
digitalWrite(13,HIGH); //acende o led
delay(500); //espera meio segundo
}

```

A chamada `pinMode(13, OUTPUT);` serve para definir que o pino 13 será uma saída. Obviamente isto só precisa ser feito no início do programa, logo está dentro de `setup()`. Se quiséssemos ter uma entrada digital, usariámos a mesma função, mas trocando `OUTPUT` por `INPUT`: `pinMode(pino,INPUT);`.

A função que define o valor de um pino digital é a `digitalWrite(pino,valor)`. Ela é chamada duas vezes no código 6.2 dentro de `loop()`, uma para apagar o led (gravando `LOW`) e outra para acendê-lo (gravando `HIGH`). `LOW` e `HIGH` são duas constantes, de valor 0 e 1, referentes ao zero e um lógico, respectivamente. Na prática, no sistema arduino, o `LOW` é uma tensão próxima a 0 V e o `HIGH` uma tensão próxima a 5 V.

Um detalhe é que o microcontrolador do arduino funciona numa velocidade de 8 ou 16 MHz (dependendo da versão), logo se colocássemos apenas as duas chamadas à função `digitalWrite` não veríamos o led piscar, mas teríamos a impressão que ele está aceso com metade da intensidade. Para vermos o led piscar é necessário colocar um atraso, que é justamente obtido pela função `delay(x)`, que gera um tempo morto de `x` milisegundos.

Se quiséssemos saber o valor de um pino digital que tivesse sido definido como entrada, a função seria `digitalRead(pino)`, que retornaria o valor digital naquele pino. Pode-se usar isto por exemplo, para fazer com que uma saída digital seja a cópia de uma entrada digital, como no código 6.3.

Código 6.3: Programa para acender um led em função de uma entrada digital.

```

const int pinoSaida = 13;
const int pinoEntrada = 10;

int valor;

void setup() {
    //codigo a ser executado no inicio
    pinMode(pinoSaida,OUTPUT); //define a saida (led)
    pinMode(pinoEntrada,INPUT); //define a entrada
}

void loop() {
    //codigo a ser executado repetidamente
    valor = digitalRead(pinoEntrada); //le a entrada
}

```

```

    digitalWrite(pinoSaida, valor); //e escreve na saída
}

```

No código 6.3 acrescentamos também algumas variáveis. Duas são constantes com os pinos usados. Elas facilitam a leitura do código e também facilitam caso posteriormente quisermos mudar os pinos utilizados. A outra variável armazena o valor lido da entrada, que depois é escrito na saída.

6.2 Sinais analógicos

Em contraste com os sinais digitais, os sinais analógicos são aqueles que podem assumir qualquer valor de tensão. No contexto do arduino, vamos por enquanto assumir que os sinais analógicos estão entre 0 V e 5 V.

Para valores analógicos, usamos as funções `analogWrite(pino,valor)` e `analogRead(pino)`, que, ao contrário das equivalentes digitais, são restritas a alguns pinos específicos. As entradas analógicas são identificadas pelos pinos ANALOG IN (A0 a A5 no arduino) e são ligadas a um conversor analógico/digital (A/D) do microcontrolador, que transforma estes sinais numa palavra binária de 10 bits. Como $2^{10} = 1024$, isto significa que a função `analogRead` retorna um valor entre 0 (para uma entrada de 0 V) e 1023 (para uma entrada de 5 V).

O arduino não tem um conversor D/A, logo a função `analogWrite` não gera um sinal analógico verdadeiro no pino. O que esta função faz é gerar um sinal modulado por largura de pulso - PWM (*Pulse Width Modulation*).

O sinal PWM é um trem de pulsos digital, com frequência da ordem de 500 Hz (no caso do arduino) cuja razão entre o tempo em alto e o período (conhecida como *duty cycle*) pode ser alterada pelo parâmetro passado, como mostra a figura 6.2. Se um sinal PWM é enviado a um pino com um led, ele piscará 500 vezes por segundo, o que é muito rápido para o olho humano, de modo que na prática o que se vê quando se varia o duty cycle de um sinal PWM que aciona um led é uma variação de sua intensidade. Logo um sinal PWM funciona, para muitas aplicações, como um sinal analógico.

Novamente, não são todos os pinos do arduino que conseguem gerar este sinal PWM, logo a função `analogWrite` está restrita aos pinos 3, 5, 6, 9, 10 e 11. Um outro detalhe que vale a pena ressaltar é que enquanto a função `analogRead` gera um valor entre 0 1023, a `analogWrite` recebe como parâmetro um valor entre 0 e 255 apenas.

Uma função útil do arduino para lidar com este tipo de situação é a função `map(valor, minIn, maxIn, minOut, maxOut)`, que faz uma transformação linear de valor de acordo com a seguinte equação:

$$(valor - minIn) \times \frac{maxOut - minOut}{maxIn - minIn} + minOut$$

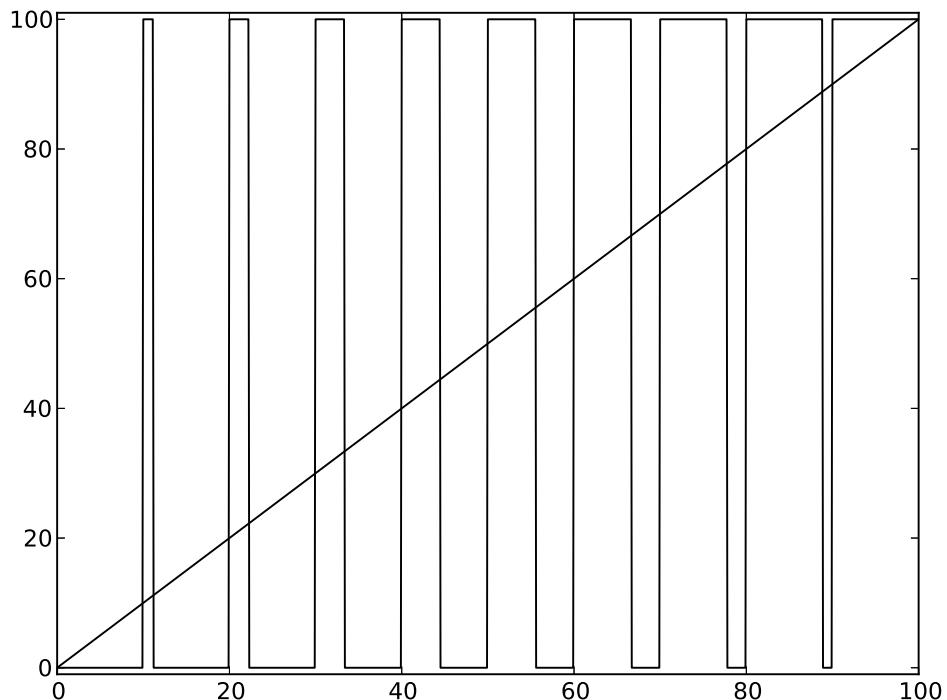


Figura 6.2: Reta modulada em largura de pulso (PWM).

A partir desta função, um código que leia o valor gerado pelo potenciômetro (em A0) e controle a intensidade do led no pino 5 poderia ser simplesmente:

```
analogWrite(5, map(analogRead(A0), 0, 1024, 0, 256));
```

Note que o valor lido pelo analogRead não precisa ser usado apenas na função analogWrite mas pode ser usado para outra finalidade, como por exemplo alterar um atraso.

6.3 Controle: for e if

Até aqui foram feitos programas puramente sequenciais, porém em vários momentos é interessante realizar operações repetidas vezes ou realizar algumas tarefas apenas em situações específicas. Para estes casos existem comandos como o **for** e o **if**. O comando for serve para tarefas repetidas. Por exemplo, se quiséssemos inicializar os pinos de 2 a 10 como saídas com valor LOW, poderíamos usar o

seguinte código:

```
for (int i = 2; i < 11; i++){
    pinMode(i,OUTPUT);
    digitalWrite(i,LOW);
}
```

O que este código faz é definir uma variável local *i* com valor inicial 2 depois ele checa se *i* é menor que 11, se for ele executa os comandos que estão entre as chaves “{” e “}”, incrementa a variável *i* (*i++*) e checa novamente. Quando *i* < 11 for falso, ele sai do laço.

O comando **if** executa um determinado código apenas se determinada condição for verdadeira. Por exemplo, para acender um led apenas se um sinal analógico for maior que a metade da escala ($512 = 1024/2$) pode-se escrever:

```
if (analogRead(A0) >= 512){
    digitalWrite(13,HIGH);
}
```

Note que este código apenas fará alguma coisa se a condição for verdadeira. Para fazer uma coisa OU outra usa-se o comando **else** após o **if**:

```
if (analogRead(A0) >= 512){
    digitalWrite(13,HIGH);
} else {
    digitalWrite(13,LOW);
}
```

6.4 Comunicação com o computador

Como já dito, a conexão USB do arduino serve também para a comunicação do mesmo com o computador. Do lado do computador o arduino aparece como uma porta serial e o próprio programa contém um terminal serial pelo qual é possível se comunicar com o arduino. Do lado do arduino, os pinos 0 e 1 são os pinos transmissor e receptor ligados ao USB .

A programação é feita através do objeto **Serial**. Este objeto tem vários comandos, porém para nós os que interessam neste momento são:

Serial.begin(baud) Inicializa a comunicação serial na velocidade (baud rate) indicada.

Serial.read() Retorna o valor de um byte recebido ou -1 caso não tenha sido recebido nenhum byte.

Serial.print(dado) e Serial.println(dado) Se dado for um char ou uma string (texto), envia dado. Se dado for um número, envia este número como uma string. No caso de println, é acrescentada uma quebra de linha após dado.

Serial.available() Retorna o número de bytes recebidos que ainda não foram lidos.

6.5 plcLib

A biblioteca plcLib (<http://www.electronics-micros.com/software-hardware/plclib-arduino/>) define um conjunto de funções muito parecidas com as da linguagem *Instruction List*, e que podem ser facilmente transcritas para a linguagem Ladder. Na configuração padrão, chamada pela função `setupPLC()` este sistema define 4 entradas (X0 a X3, usando A0 a A3) e quatro saídas (Y0 a Y3, usando 3, 5, 6 e 9).

Usando o Multi-function Shield, as entradas correspondem ao potenciômetro (A0) e aos 3 botões, logo estão bem adequadas. As saídas correspondem ao buzzer (3) e a três conectores de servo, porém não temos nenhuma saída ligada aos leds. Para definir todas entradas e saídas de uma vez, fazemos:

Código 6.4: Definição de função e constantes para usar o o Multi-function Shield com o plcLib.

```
// Definicao das constantes dos pinos
const int VR = A0;
const int S1 = A1;
const int S2 = A2;
const int S3 = A3;
const int D1 = 13;
const int D2 = 12;
const int D3 = 11;
const int D4 = 10;
const int LS1 = 3;
const int Q1 = 5;
const int Q2 = 6;
const int Q3 = 9;
const int Q4 = A5;

void setupShield(){ // Funcao para inicializar
    pinMode(VR,INPUT);
    pinMode(S1,INPUT);
    pinMode(S2,INPUT);
    pinMode(S3,INPUT);
    pinMode(D1, OUTPUT);
    pinMode(D2, OUTPUT);
    pinMode(D3, OUTPUT);
    pinMode(D4, OUTPUT);
    pinMode(Q1, OUTPUT);
    pinMode(Q2, OUTPUT);
    pinMode(Q3, OUTPUT);
    pinMode(Q4, OUTPUT);
```

```

pinMode(LS1, OUTPUT);
// Valores iniciais para apagar os leds e nao soar a buzina
digitalWrite(D1,HIGH);
digitalWrite(D2,HIGH);
digitalWrite(D3,HIGH);
digitalWrite(D4,HIGH);
digitalWrite(LS1,HIGH);
}

```

Deste modo basta chamarmos `setupShield()` no `setup` que configuramos nossa entradas e saídas.

6.5.1 Acumulador

Os CLPs trabalham com o conceito de acumulador (que na implementação do `plcLib` tem o nome `scanValue`), que é uma variável implícita. As funções pegam implicitamente o valor a ser trabalhado do acumulador e salvam o resultado de volta nele. O `plcLib` define as funções `in(entrada)`, `inNot(entrada)`, `out(saida)` e `outNot(saida)`, equivalentes às LD, LDN, ST e STN, de Instruction List, respectivamente. As duas primeiras pegam o valor de uma entrada e guardam no acumulador, enquanto que as 2 últimas pegam o valor do acumulador e colocam na saída. O Not (N) no final indica que estes valores são invertidos. Vejamos como exemplo o código 6.5:

Código 6.5: Código simples de leitura de cópia de entradas para saídas.

```

void loop() {
    in(S1);      // Le chave 1
    out(D1);     // manda para led 1

    inNot(S2);   // Le chave 2 (invertida)
    out(D2);     // manda para led 2

    inNot(S3);   // Le chave 3 (invertida)
    outNot(D3);  // manda para led 2 (invertido)
}

```

Neste caso, D1 fica sendo uma cópia de S1, D2 o valor invertido de S2. Dá para imaginar o que acontece com D3.

É bom lembrar que no multi-function shield as chaves são conectadas ao terra, enquanto que os leds são conectados à 5V. Isto faz com que quando apertada, a chave gere um 0 lógico (não um 1) e que o led acende quando se envia um 0 lógico (e não um 1). Por isto é interessante para trabalharmos com este shield fazermos

`inNot(Sn)`, que resulta em 1 se a chave estiver apertada, e `outNot(Ln)`, que acende o led quando o valor no acumulador for 1.

6.5.2 Funções lógicas

Mas apenas carregar um único valor não é tão interessante. Bem mais útil é montar uma relação lógica entre valores. Para isto servem as funções `andBit`, `orBit`, `xorBit`, `andNotBit` e `orNotBit`. Cada uma destas funções faz uma operação lógica entre o acumulador e a variável indicada, salvando o resultado no acumulador. A tabela 6.1 mostra as tabelas verdadeas de cada uma delas.

Tabela 6.1: Tabela verdade das funções lógicas definidas em `plcLib`.

scanValue	ent	scanValue (após operação)				
		andBit	orBit	xorBit	andNotBit	orNotBit
0	0	0	0	0	0	1
0	1	0	1	1	0	0
1	0	0	1	1	1	1
1	1	1	1	0	0	1

Colocando estas funções em cascata é possível criar lógicas bem interessantes, como por exemplo, soar o alarme se S1 ou S2 forem pressionados ao mesmo tempo que S3 for pressionado.

Código 6.6: Aciona a buzina em função das chaves.

```
inNot(S1); // Le se chave 1 apertada
orNotBit(S2); // Le se chave 2 apertada e faz um OU logico
andNotBit(S3); // Le se chave 3 apertada e faz um E logico
outNot(LS1); // Se condicao satisfeita , aciona a buzina
```

O problema do uso do acumulador é que às vezes uma situação exige uma lógica mais complexa do que o acumulador permite. Por exemplo: como acender D2 se a maioria dos botões estiver pressionado? (Problema do voto majoritário) Existem 2 formas de resolver este problema: pilha ou variáveis;

A pilha é como a maioria dos CLPs trabalham: ao invés de ter um único acumulador, ele tem um número finito de posições de memória que ele usa e todo comando usa implicitamente o topo da pilha. No caso da implementação do `plcLib`, a pilha pode ser definida separadamente como um objeto da classe `Stack`. Logo, se fizermos `Stack pilha`; definimos uma pilha de nome `pilha`.

Os principais comandos de acesso à pilha são o `push()`, que passam o valor do acumulador para o topo da pilha, e o `pop()`, que tira o valor do topo da pilha e passa para o acumulador. Outros comandos úteis para a lógica são o `andBlock()`,

que faz o E do valor no topo da pilha com o acumulador, salvando no acumulador e o orBlock(), que faz a mesma coisa com o OU lógico. Usando a pilha é possível resolver o problema da maioria dos botões da seguinte forma:

Código 6.7: Solução do voto majoritário com uso de pilha.

```
void loop(){\lstinline|
    inNot(S1);
    andNotBit(S2);
    andBit(S3);
    pilha.push();
    inNot(S1);
    andBit(S2);
    andNotBit(S3);
    pilha.push();
    in(S1);
    andNotBit(S2);
    andNotBit(S3);
    pilha.push();
    inNot(S1);
    andNotBit(S2);
    andNotBit(S3);
    pilha.orBlock();
    pilha.orBlock();
    pilha.orBlock();
    outNot(D2);
}
```

Outra possibilidade é simplesmente definir variáveis auxiliares para receberem os valores intermediários. Neste caso as variáveis precisam ser do tipo **unsigned int**.

Código 6.8: Solução do voto majoritário com uso de variável.

```
int temp1;
void loop(){
    inNot(S1);
    andNotBit(S2);
    andBit(S3);
    out(temp1);
    inNot(S1);
    andBit(S2);
    andNotBit(S3);
    orBit(temp1);
    out(temp1);
    in(S1);
    andNotBit(S2);
```

```

    andNotBit(S3);
    orBit(temp1);
    out(temp1);
    inNot(S1);
    andNotBit(S2);
    andNotBit(S3);
    pilha.orBlock();
    pilha.orBlock();
    pilha.orBlock();
    orBit(temp1);
    outNot(D1);
}

```

Note-se que não se usou a melhor estratégia lógica para ambas soluções apresentadas. Fica como exercício resolver este problema de forma mais compacta.

6.5.3 Memória

Até o momento usamos apenas a chamada lógica combinacional, onde a saída depende apenas da entrada. Porém é bem comum a situação de querermos que a saída fique num determinado estado em função da sua história pregressa.

Um exemplo simples: Como acionar LS1 apertando S1 e pará-lo apertando S2? O estado de LS1 depende então de qual foi o último botão apertado.

É possível implementar este tipo de memória a partir de lógica combinacional usando realimentação. Ou seja, devemos ler a saída do sistema como sendo entrada, o que, apesar de estranho, é perfeitamente possível. O código 6.9 mostra justamente esta implementação.

Código 6.9: Implementação de latch com lógica combinacional.

```

inNot(S1);      // Se chave S1 apertada
orNotBit(LS1); // ou se LS1 ja esta acionado
andBit(S2);    // mas apenas se S2 nao estiver apertada
outNot(LS1);   // aciona LS1

```

Como esta função é bastante utilizada, ela recebeu o nome de *latch* (tranca, ferrolho) e tem funções específicas, para deixar uma variável em 1 (função `set()`) ou 0 (`reset`) dali em diante, como mostrado no código 6.10.

Código 6.10: Funções para uso de latch.

```

inNot(S1);      // Se chave S1 apertada
reset(LS1);    // Seta LS1 (aciona dai em diante)
andBit(S2);    // Se S2 estiver apertada
set(LS1);      // Reseta LS1 (desliga dai em diante)

```

6.5.4 Temporizadores

O arduino já tem uma função padrão delay, que causa a paralisação da execução por um determinado tempo. O problema de delay é que ele para o programa todo, o que pode ocasionar problemas até de segurança.

Os temporizadores definidos na plcLib permitem gerarmos atrasos em sinais específicos, sem mexer nos demais. O timerOn atrasa a subida de um sinal, enquanto que o timerOff atrasa a descida de um sinal, como pode ser visto no exemplo abaixo. Outro tipo de temporizador é o timerPulse, que na subida do sinal de entrada gera um pulso por um tempo determinado. timerPulse e timerOff são bem parecidos, com a diferença que o último começa a medir o tempo a partir da descida da entrada, enquanto que o primeiro mede a partir da subida da entrada. Cada função destas recebe como parâmetro a variável (do tipo **unsigned long**) que contará o tempo e o tempo final a ser contado. Logo para cada temporizador é necessário ter uma variável para armazenar o tempo utilizado.

Código 6.11: Exemplos de uso de temporizadores.

```
unsigned long TIMER0 = 0;
unsigned long TIMER1 = 0;
unsigned long TIMEa = 0;
void loop(){
    inNot(S1);      // Se chave S1 apertada
    timerOn(TIMER0,2000); // atrasa subida por 2 segundos
    outNot(D1);
    inNot(S2);      // Se chave S2 apertada
    timerOff(TIMER1,2000); // atrasa descida por 2 segundos
    outNot(D2);
    inNot(S3);      // Se chave S3 apertada
    timerPulse(TIMEa, 2000); // gera pulso de 2 segundos
    outNot(D3);
}
```

Um outro temporizador interessante é o timerCycle, que enquanto o acumulador for 1, gera um sinal alternado definido por 2 tempos: o tempo em alto e o tempo em baixo. Muito útil para alarmes.

Código 6.12: Exemplos de uso timerCycle.

```
unsigned long TempoLOW = 0;
unsigned long TempoHIGH = 0;
void loop(){
    inNot(S1);
    timerCycle(TempoLOW,1300 ,TempoHIGH,100 );
    outNot(LS1);
```

}

6.5.5 Contadores

Um contador incrementa ou decremente uma variável de acordo com o número de eventos que recebe. Na implementação do plcLib, estes eventos são subidas (ir de 0 para 1) em suas entradas.

Para os contadores, cria-se um objeto do tipo Counter, que tem um valor interno de presetValue, uma variável de contagem count, 4 entradas countUp, countDown, preset, clear e 2 saídas upperQ e lowerQ. As regras do contador são:

- Uma subida em countUp incrementa o valor de count, se for menor que preset.
- Uma subida em countDown decrementa o valor de count, se for maior que 0.
- Uma subida em clear faz count igual a 0.
- Uma subida em preset faz count igual a presetValue.
- Se count for igual a 0, lowerQ retorna 1.
- se count for igual a preset, upperQ retorna 1.

Código 6.13: Exemplo de uso do contador.

```
Counter ctr(10);           // Contador na faixa 0–10, começando em zero
unsigned long TIMER0 = 0;   //
unsigned long TIMER1 = 0;   //

void loop() {
    inNot(S1);           // Se S1 apertada
    timerOn(TIMER0, 10); // 10 ms de atraso para debounce
    ctr.countUp();        // incrementa ctr.
    inNot(S2);           // Se S2 apertada
    timerOn(TIMER1, 10); // 10 ms de atraso para debounce
    ctr.countDown();     // decrementa ctr.
    inNot(S3);           // Se S3 apertada
    ctr.clear();          // zera o contador.
    ctr.lowerQ();         // Se zero,
    outNot(D1);           // Se 10,
    ctr.upperQ();         // Se 10,
    outNot(D2);
}
```

É possível ainda criar um contador que inicializa no valor de preset. Tomando o exemplo do código 6.13, ao invés de fazer Counter ctr(10);, faria-se Counter ctr(10,1);

6.6 Linguagem Ladder – Lógica Booleana

6.7 De relês a CLPs

Antes do desenvolvimento de CLPs, a lógica de operação de sistemas industriais era feita através de dispositivos eletromecânicos chamados relês, que são que chaves controladas eletricamente. Ainda hoje há várias aplicações que utilizam relês, principalmente quando se trabalha com altas tensões e correntes e em baixa velocidade. Um elemento muito usado na indústria é o contator, que nada mais é do que um relê trifásico.

Um relê eletromecânico é um chave que se mantém em uma determinada posição (fechada ou aberta) pela ação de uma mola e que muda de estado (abre ou fecha) pela ação de um eletroímã. Ou seja, ao se magnetizar este eletroímã o relê abre (fecha) e volta a fechar (abrir) sem esta magnetização. Eletricamente pode-se visualizar um relê apenas como um indutor e uma chave, e muitas vezes é desta forma que ele é representado em um diagrama esquemático (vide exemplo na Figura 6.3(b)).

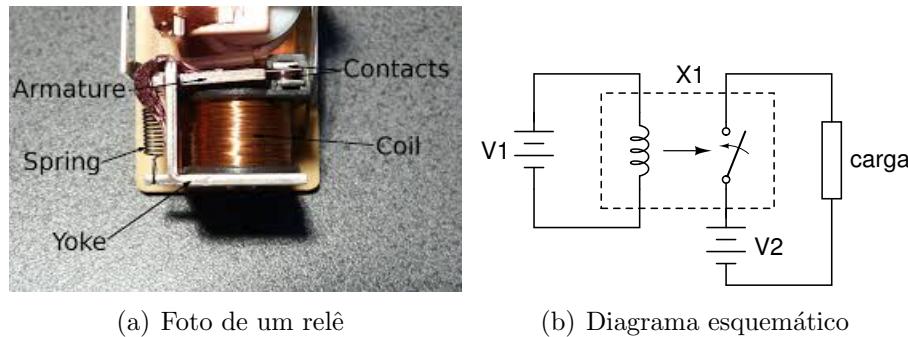


Figura 6.3: Foto de um relê e representação esquemática.

O estado de repouso define o tipo básico do relê: normalmente aberto ou normalmente fechado. Também é comum relês que tenham 2 ou mais chaves acionadas por uma mesma bobina, inclusive podendo ser uma normalmente aberta e outra normalmente fechada, tal como o relê da figura 6.3(a).

A parte de acionamento de um relê (a bobina) é eletricamente isolada do contato. Esta característica é interessante para separar o circuito de controle do circuito de acionamento, o que permite o acionamento de cargas de alta tensão e alta corrente a partir de um circuito de baixa tensão. Um exemplo de aplicação de relê é mostrado na figura 6.4, onde 2 relês c1 e c2 são usados para o acionamento de um motor trifásico em comandado pelas batoeiras b0, b1 e b2.

Em geral é mais fácil analisar um circuito do ponto de vista lógico separando a parte de acionamento da parte de controle, então é comum que o símbolo de um relê seja separado em duas partes: a bobina (o eletroímã) e o contato (a chave), interligados pelo mesmo nome. Isto é exemplificado na figura 6.5, que redesenha

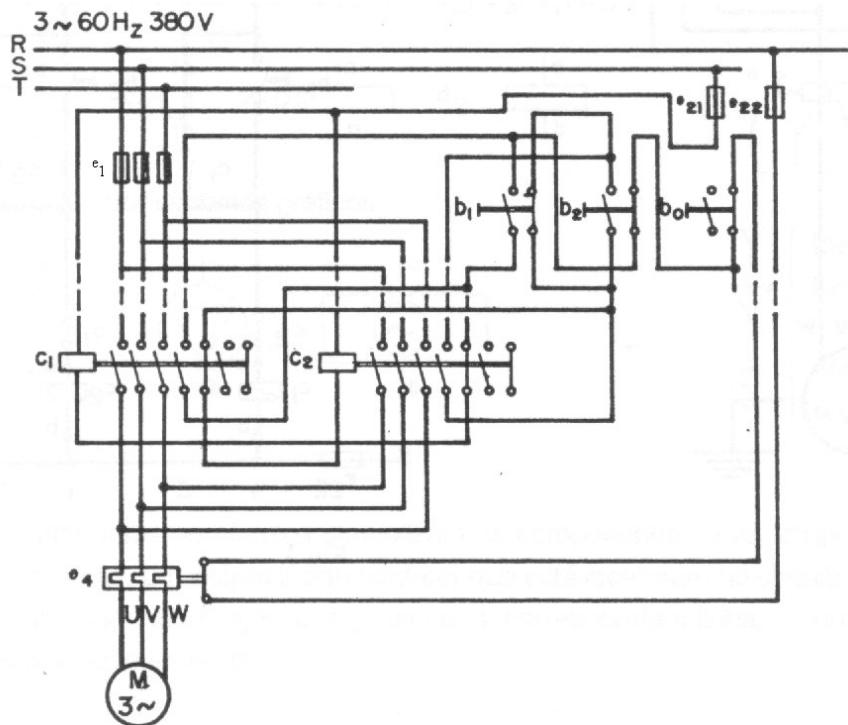


Figura 6.4: Circuito de acionamento de um motor trifásico por relê.

o mesmo circuito da figura 6.4 separando a parte de controle da de acionamento, tornando o circuito bem menos convoluto. A ligação entre os diversos contatos e bobinas dos relês permite a realização de diversas funções interessantes para o controle de circuito. Tais circuitos ficaram conhecidos por *circuitos chaveados*.

6.7.1 Diagrama Ladder

Os diagramas ladder são muito utilizados para representar circuitos com relês enfatizando a lógica da ligação. Neste tipo de diagrama as bobinas tem o símbolo $\textcircled{-}$, os contatos normalmente abertos são simbolizados por $\textcircled{+}$ e os normalmente fechados por $\textcircled{\times}$.

A Figura 6.6 mostra o mesmo circuito de controle da Figura 6.5, só que agora descrito em ladder. Com os circuitos conectados desta maneira, o diagrama fica parecendo uma escada; daí o nome¹.

Logo de cara nota-se que não existem neste diagrama as tensões V1 e V2. Isto se dá pois neste diagrama considera-se que as tensões estão entre as duas barras

¹em inglês *ladder* significa escada.

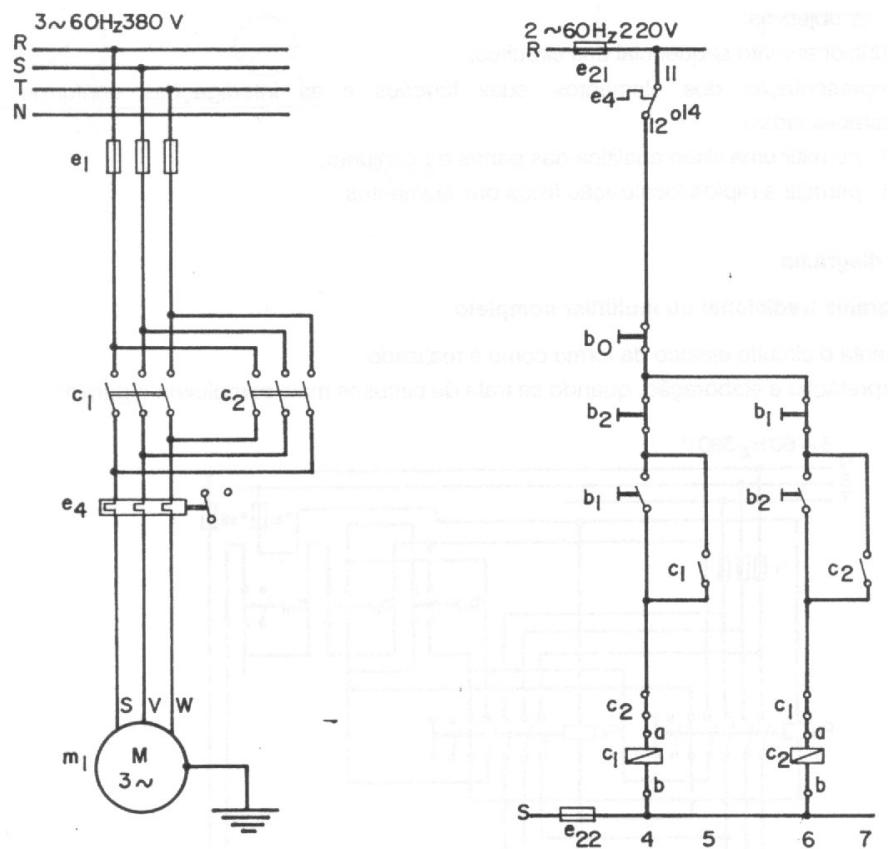


Figura 6.5: Mesmo circuito da figura 6.4, separando a parte de controle da de acionamento.

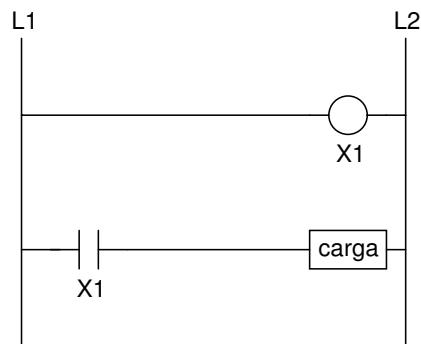
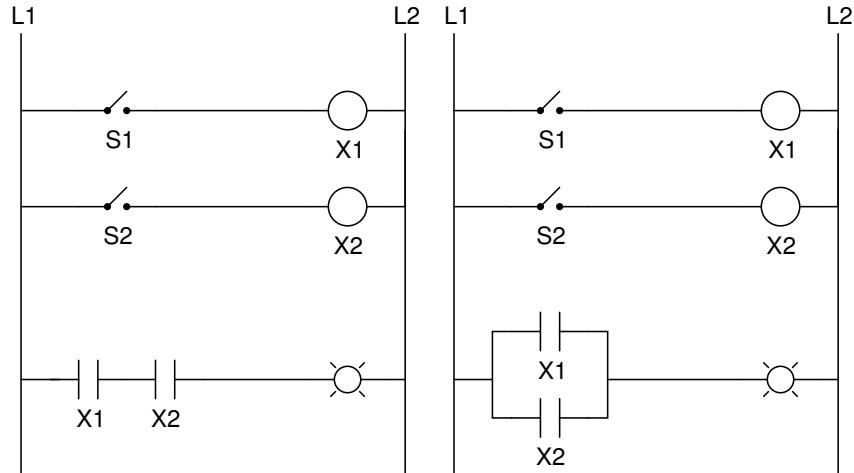


Figura 6.6: Exemplo de circuito de controle em ladder.

L_1 e L_2 e abstrai-se a fonte de tensão. Isto lembra, de certo modo, uma ligação real, com os elementos conectados entre os cabos de fase e o neutro.

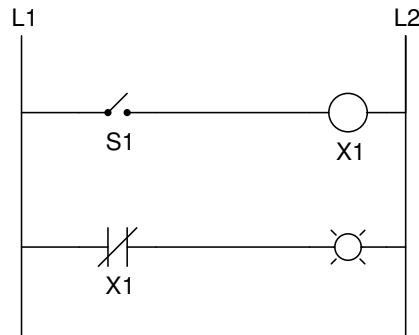
Com este tipo de diagrama, fica fácil analisar a lógica por trás dos circuitos, o que fez com que este diagrama ainda seja usado, só que agora como uma linguagem gráfica de programação de CLPs.

Como exemplo, a figura 6.7 mostra diagramas ladder que acendem uma lâmpada apenas quando 2 relês são acionados, se qualquer um dos relês for acionado ou quando nenhum dos relês é acionado.



(a) Aciona apenas se ambos relês estiverem acionados.

(b) Aciona se um ou outro relê estiver acionado.



(c) Aciona se o relê não estiver acionado.

Figura 6.7: Exemplos de circuitos lógicos chaveados em ladder.

O engenheiro Claude Shannon descobriu em 1934 a relação entre os circuitos chaveados e a álgebra de Boole, que é um mapeamento da lógica em uma álgebra. Uma ligação em série realiza um AND lógico (uma multiplicação booleana), uma ligação paralela realiza um OR lógico (uma soma na álgebra de Boole) e o uso de um contato normalmente fechado realiza a inversão lógica, ou o NOT (representado por uma barra sobre a variável). A álgebra de Boole mostra que a

combinação destas três operações, e portanto a combinação destes três circuitos, permite realizar qualquer condição lógica para o acionamento do que quer que seja. Porém a aplicação direta dos postulados e teoremas desta álgebra é muitas vezes não intuitiva e portanto complicada e passível a erros.

6.8 Mapas de Karnaugh

O chamado mapa de Karnaugh foi desenvolvido pelo matemático e físico Maurice Karnaugh em 1953, enquanto trabalhava no grupo de pesquisas da empresa Bell. Este método é uma poderosa ferramenta para circuitos lógicos.

Como as equações da álgebra de Boole tratam, também, de probabilidade, elas podem ser visualizadas através de um diagrama de Venn. Isto é exemplificado na figura 6.8, que apresenta um diagrama de Venn de 3 variáveis com todas as regiões demarcadas.

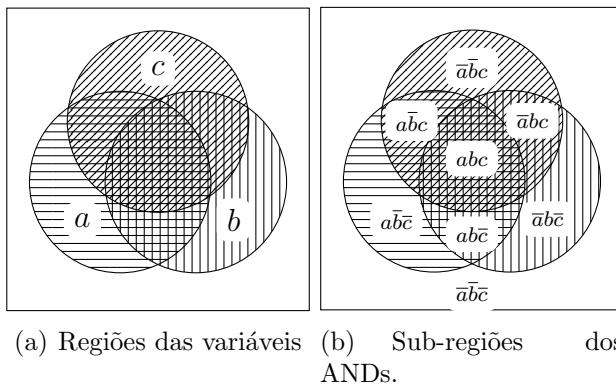


Figura 6.8: Diagrama Venn com 3 variáveis.

Utilizando os diagramas, é fácil obter a equação simplificada da função. Por exemplo, considere-se a função $f_1 = \bar{a}\bar{b}c + \bar{a}bc + \bar{a}\bar{b}\bar{c}$. Desenhando esta função num diagrama de Venn (figura 6.9), fica óbvio que podemos simplificá-la para $f_1 = (a + c)\bar{b}$.

O problema aparece quando acrescentamos mais 1 variável. Como fazer um diagrama definindo todas as 16 possibilidades? Uma solução para isto é desenhar as regiões como retângulos e não como círculos, assim como foi feito na figura 6.10, lado esquerdo. Uma melhora é ainda aplicada no diagrama do lado direito desta figura, onde a indicação de que regiões correspondem a que variáveis é feita não pelo padrão da área, mas sim indicada no lado externo.

O mapa de Karnaugh já é este diagrama de Venn modificado, onde o resultado da função booleana mapeada é marcado em cada região (casa). Cada casa em um mapa de Karnaugh corresponde a uma linha na tabela verdade, que é um AND

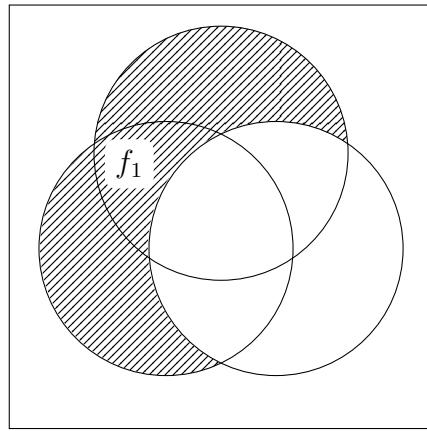


Figura 6.9: Diagrama Venn definindo a região dada por $f_1 = a\bar{b}\bar{c} + a\bar{b}c + \bar{a}\bar{b}c = (a + c)\bar{b}$.

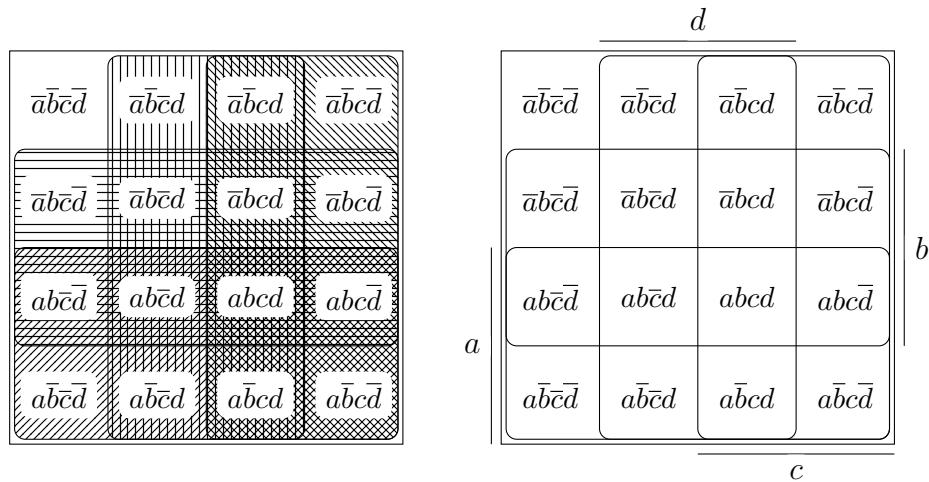


Figura 6.10: Diagrama Venn de 4 variáveis desenhado com regiões quadradas.

de todas as variáveis envolvidas – vamos chamar isto de mintermo. A figura 6.11 mostra os mintermos correspondentes a cada uma das regiões.

Note na figura 6.11 que os vizinhos de cada casa em um mapa de Karnaugh são tais que apenas muda uma variável de cada vez. Por exemplo, da casa 5 para a 1 (acima) só muda o b , da 5 para a 7 (direita) só muda o c , da 5 para a 4 (esquerda) só muda o d e da 5 para a 13 (abaixo) só muda o a .

Isto que foi mostrado para a casa 5 é válido para todas casas, inclusive para as bordas e quinas, pois podemos considerar que o mapa dá a volta em si mesmo. Deste modo considera-se a casa 6 como vizinha da 4 e só muda a variável c , a casa 10 vizinha da 2 e só muda a variável a e assim por diante.

d				
$\bar{a}\bar{b}\bar{c}\bar{d}$	$\bar{a}\bar{b}cd$	$\bar{a}bc\bar{d}$	$\bar{a}bcd$	
a	$\bar{a}b\bar{c}\bar{d}$	$\bar{a}b\bar{c}d$	$\bar{a}bcd$	$\bar{a}bcd$
	$a\bar{b}\bar{c}\bar{d}$	$a\bar{b}cd$	$ab\bar{c}\bar{d}$	$ab\bar{c}d$
	$a\bar{b}\bar{c}\bar{d}$	$a\bar{b}cd$	$ab\bar{c}\bar{d}$	$ab\bar{c}d$

Figura 6.11: Mapa de Karnaugh com os mintermos correspondentes a cada casa.

Desta característica do mapa de Karnaugh vem sua principal utilidade. Por exemplo, considere a função $f_2 = \Sigma_m(3, 7, 12, 13)$ – o somatório dos mintermos das casas 3, 7, 12 e 13 – e seu respectivo mapa de karnaugh na figura 6.12.

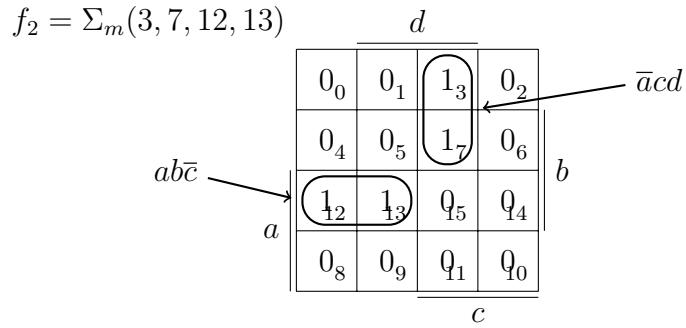


Figura 6.12: Função f_2 e simplificação por agrupamento de casas vizinhas.

Analizando a função f_2 por álgebra de Boole, vemos que podemos simplificá-la através da aplicação do teorema que diz que $\bar{a}\bar{b} + ab = a$ e, observando no mapa de Karnaugh, os termos que são unidos e simplificados são justamente os vizinhos.

$$f_2 = \underbrace{\bar{a}\bar{b}cd + \bar{a}bcd}_{\bar{a}cd} + \underbrace{ab\bar{c}\bar{d} + ab\bar{c}d}_{ab\bar{c}}$$

Ou seja, o agrupamento de 2 casas vizinhas corresponde à simplificação de uma variável. Basta ver no próprio mapa quais são as variáveis que não mudam dentro do agrupamento.

Para simplificar 2 ou mais variáveis basta aplicar o teorema repetidas vezes. Simplifiquemos a função f_3 (vide figura 6.13), por exemplo. Basta agruparmos a função de duas em duas casas e 2 grupos vizinhos de duas casas viram um único grupo de 4 casas, retirando mais uma variável da função.

$$\begin{aligned} f_3 &= \underbrace{a\bar{b}\bar{c}\bar{d}}_{ab\bar{c}} + \underbrace{a\bar{b}\bar{c}d}_{ab\bar{c}} + \underbrace{ab\bar{c}\bar{d}}_{a\bar{b}\bar{c}} + \underbrace{ab\bar{c}d}_{a\bar{b}\bar{c}} \\ &= \underbrace{a\bar{b}\bar{c}}_{f_3} + ab\bar{c} \end{aligned}$$

$$f_3 = a\bar{c}$$

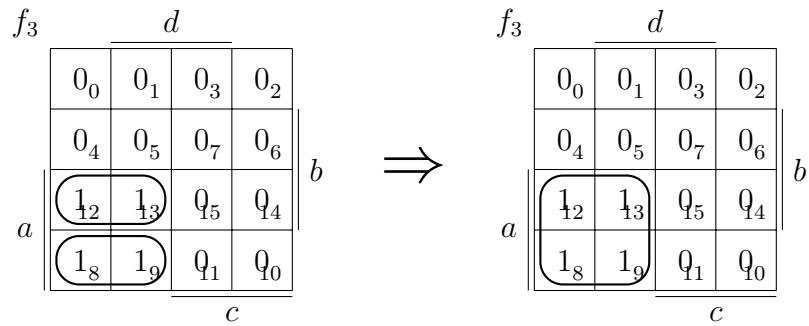


Figura 6.13: Agrupamento das casas da função f_3 .

Este mesmo procedimento pode ser mostrado para agrupamentos de 8 casas (simplificando então 3 variáveis) ou 16 casas (simplificando 4 variáveis. A figura 6.14² mostra algumas possibilidades de agrupamentos de 2, 4 e 8 casas, junto com o produto respectivo. Num mapa de Karnaugh de 4 variáveis um agrupamento de 16 casas seria todo o mapa e corresponderia a função 1.

²Exemplos retirados do artigo original de Karnaugh: “The map method for synthesis of combinational logic circuits”, de 1953.

$b\bar{c}d$	d	$\bar{a}\bar{b}d$	d	$\bar{a}\bar{b}\bar{d}$	d	$\bar{b}\bar{c}\bar{d}$	d
a	$0\boxed{0}000$	b	$0\boxed{1}D00$	b	$D\boxed{0}001$	b	$\boxed{1}0000$
	$0\boxed{1}000$	a	$00\boxed{0}00$	a	$000\boxed{0}0$	a	$000\boxed{0}0$
	$00\boxed{1}00$		$0000\boxed{0}$		$0000\boxed{0}$		$0000\boxed{1}$
	$00000\boxed{0}$				$00000\boxed{0}$		
					$00000\boxed{1}$		
					$0000\boxed{1}0$		
					$000\boxed{1}00$		
					$0\boxed{1}000$		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		
					00001		
					00010		
					00100		
					01000		
					10000		

6.8.1 Mapas de n variáveis

É fácil fazer um mapa de Karnaugh com um número menor de variáveis (i.e.: $n < 4$). Para tanto basta simplesmente sair dividindo o mapa. Deve-se apenas lembrar que uma casa deve ter n vizinhas, já que a simplificação de uma variável corresponde a unir uma casa com a vizinha. Isto é mostrado na figura 6.15 para mapas de 2, 3 e 4 variáveis.

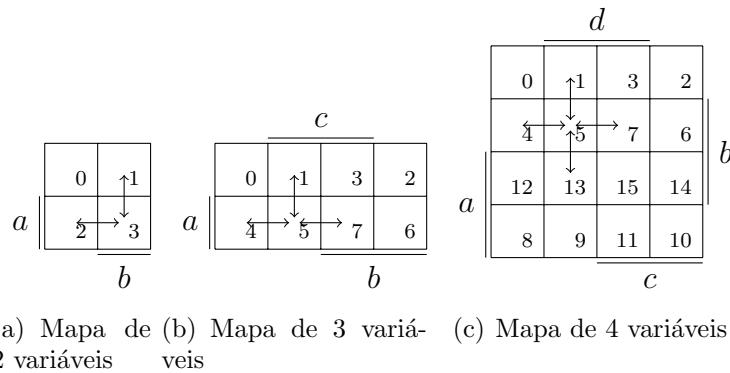


Figura 6.15: Mapas de Karnaugh de $n = 2, 3, 4$ variáveis, mostrando que cada casa tem n vizinhos.

Mas como aplicar este princípio para funções com mais de 4 variáveis? É impossível fazer um mapa no plano onde cada uma das regiões tem 5 (ou mais) vizinhos. Uma maneira (não muito prática) é trabalhar com um mapa tridimensional como exemplifica a figura 6.16 que mostra um mapa de Karnaugh de 6 variáveis, note que cada casa tem 6 vizinhos: 4 no plano (como no mapa de 4 variáveis) e 2 verticais.

Na prática, um mapa de 5 variáveis é desenhado como 2 de 4 variáveis, sendo um com uma variável (em geral a mais significativa) sendo 0 e o outro com a mesma variável sendo 1. Usa-se este mesmo princípio para mapas de 6 ou mais variáveis, como pode ser visto na figura 6.17.

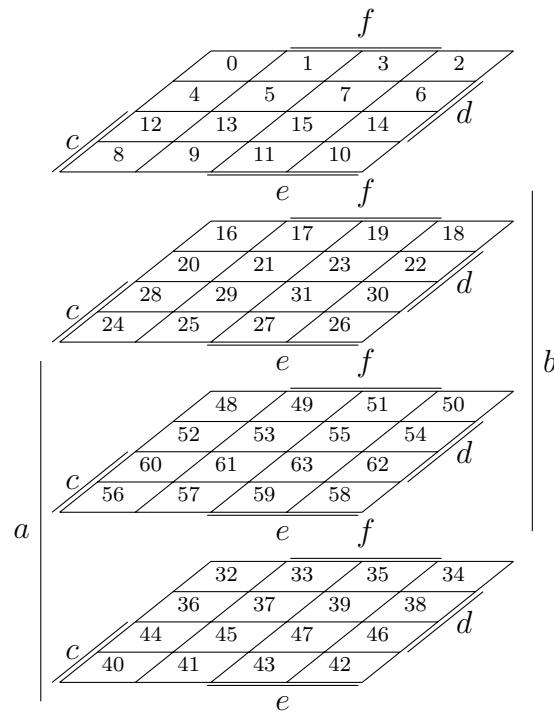


Figura 6.16: Mapa de Karnaugh tridimensional de 6 variáveis.

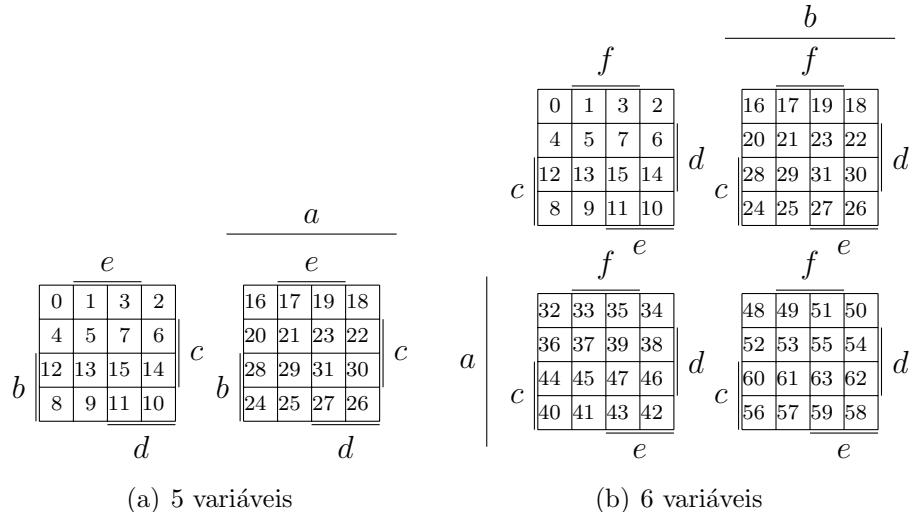


Figura 6.17: Mapas de Karnaugh de 5 e 6 variáveis.

6.9 Equações e circuitos não-completamente especificados.

É bastante comum a situação de que determinadas entradas de um circuito lógico nunca ocorram. Como exemplo imagine-se uma esteira carregando uma caixa de um lado para o outro, com sensores de fim de curso em ambas extremidades: s_e do lado esquerdo e s_d do lado direito. No funcionamento normal do sistema estes dois sinais nunca serão acionados ao mesmo tempo; nesta situação não importa qual é o resultado do circuito para $s_e = s_d = 1$, já que esta situação nunca vai existir. Diz-se então que este circuito é não-completamente especificado.

A tabela 6.2 mostra o exemplo de um sinal imaginário z determinado em função de a , s_e e s_d . Neste exemplo, sempre que $s_e = s_d = 1$ a saída z é não-especificada, ou seja, z *não-importa* nestas situações. Neste texto utilizaremos a notação ‘ \times ’ para identificar as situações que um sinal não importa. Outras notações comumente usadas são ‘ $*$ ’, ‘ $-$ ’ ou ‘ d ’.

Tabela 6.2: Exemplo de uma função não completamente especificada.

a	s_e	s_d	z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	\times
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	\times

Pode-se descrever uma função lógica não-completamente especificada na forma soma de mintermos utilizando a notação $d(\dots)$, que vem do inglês *don't care*. Desta forma o sinal z pode ser descrito por:

$$z = \Sigma_m(2, 4, 6) + d(3, 7) \quad (6.1)$$

Um \times na saída pode ser implementado como um 1 ou um 0, e não se sabe a princípio qual destes dois valores gerará uma solução mais minimizada, logo para obter o menor circuito possível o engenheiro deveria, a princípio, obter as equações considerando que cada \times pode ser 1 ou 0 e checar qual é o menor circuito final. Obviamente para problemas com muitos \times 's isto se torna impraticável, pois seria necessário minimizar 2^k funções, onde k é o número de \times 's presentes.

O mapa de Karnaugh facilita bastante a implementação de circuitos não-completamente especificados, pois podemos considerar se determinado x é 1 ou 0 visualmente, na hora da implementação.

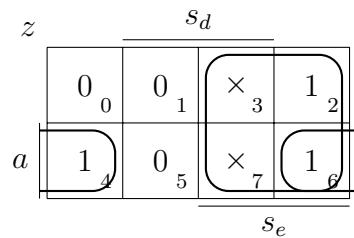


Figura 6.18: Mapa de Karnaugh da função z .

6.10 Linguagem Ladder – Temporizadores

6.11 Linguagem Ladder – Contadores

6.12 Linguagem Ladder – Aplicações

Capítulo 7

Linguagem Grafset

O Grafset pode ser definido em 2 níveis de abstração: o 1 e o 2. O nível 1 serve como uma ferramenta de desenvolvimento, para analisar a partir do problema como um todo a sequência de etapas, as ações a serem realizadas em cada etapa e as condições de transição de uma etapa para outra. No nível 1 as ações e transições são descritas de forma ampla, para permitirem uma melhor visualização do processo pelo desenvolvedor. O resultado final é uma descrição dos requisitos daquelas etapas e transições e não serve como uma linguagem para programar um sistema.

Já o Grafset nível 2 é propriamente uma linguagem de programação, com diferentes implementações. Uma delas é o SFC - *Sequential Function Chart*, descrita no padrão IEC1131-3 para programação de CLPs. No SFC, cada ação corresponde a uma atuação nas saídas ou variáveis internas do CLP e cada transição corresponde a um valor binário obtido no CLP seja de entradas, seja de comparações de valores analógicos ou de tempo.

Desta forma o principal uso do grafset é num projeto top-down, onde a partir do problema se desenvolve a sequência a ser seguida no grafset nível 1, a partir deste se definem todas as entradas, saídas e condições necessárias para o controle automático daquela sequência e então programa-se o sistema usando grafset nível 2.

Tomemos como exemplo uma tarefa relativamente simples: cozinhar um ovo.

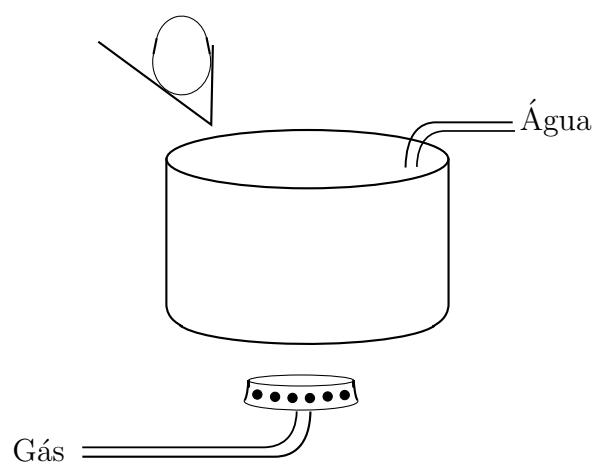


Figura 7.1: Sistema para cozinhar ovo sem sensores ou atuadores.

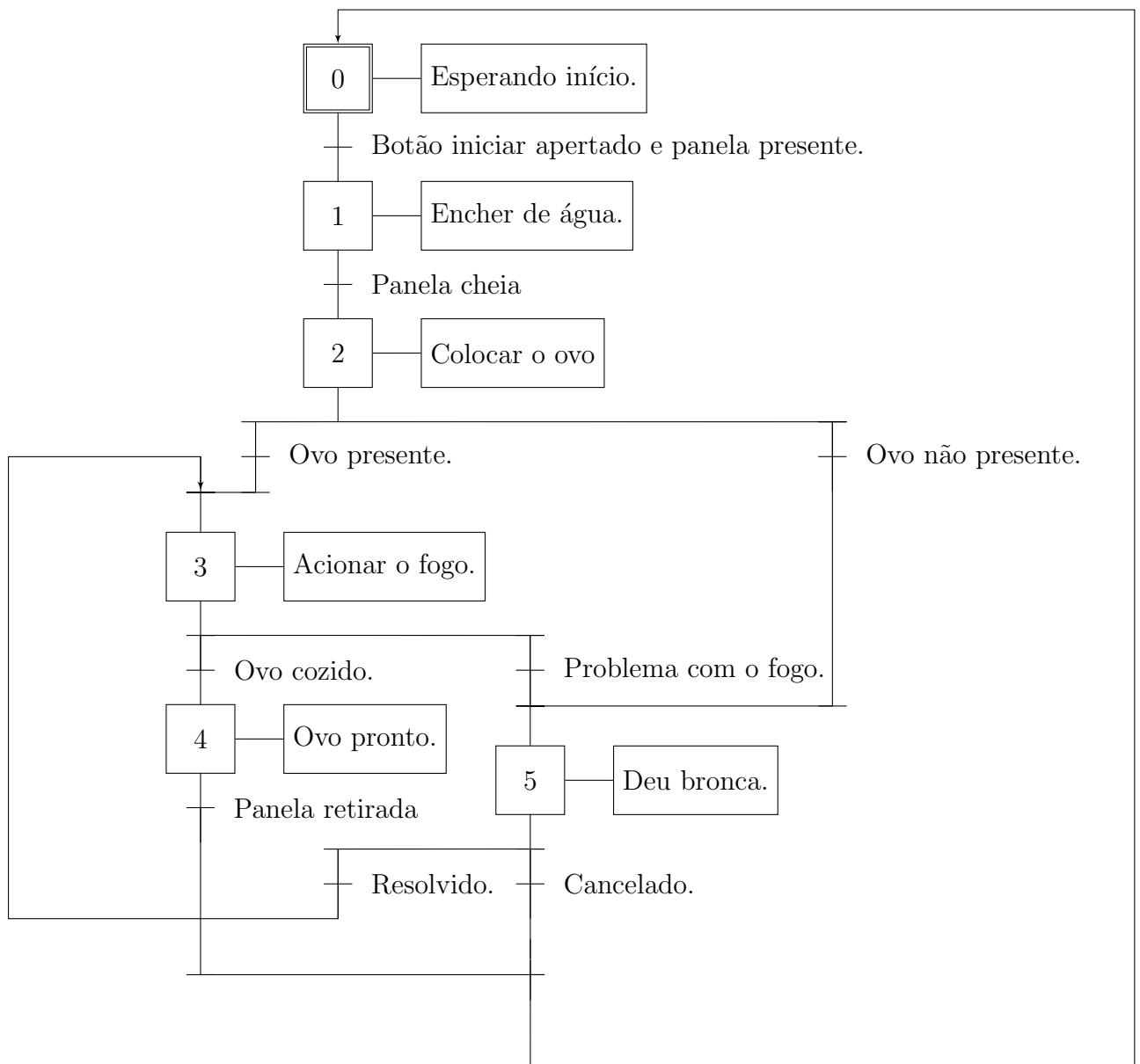


Figura 7.2: Diagrama grafcet nível 1 para cozinhar um ovo.

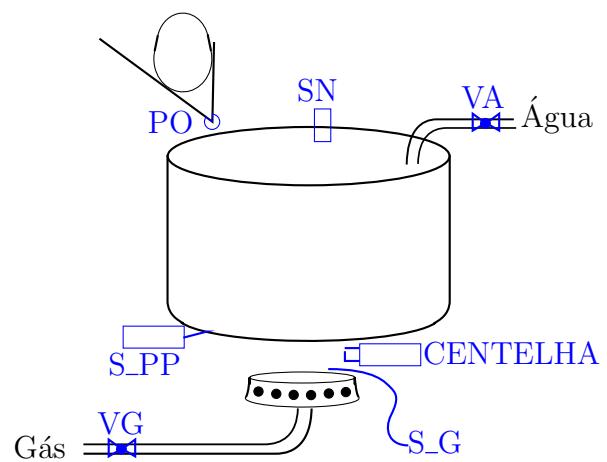
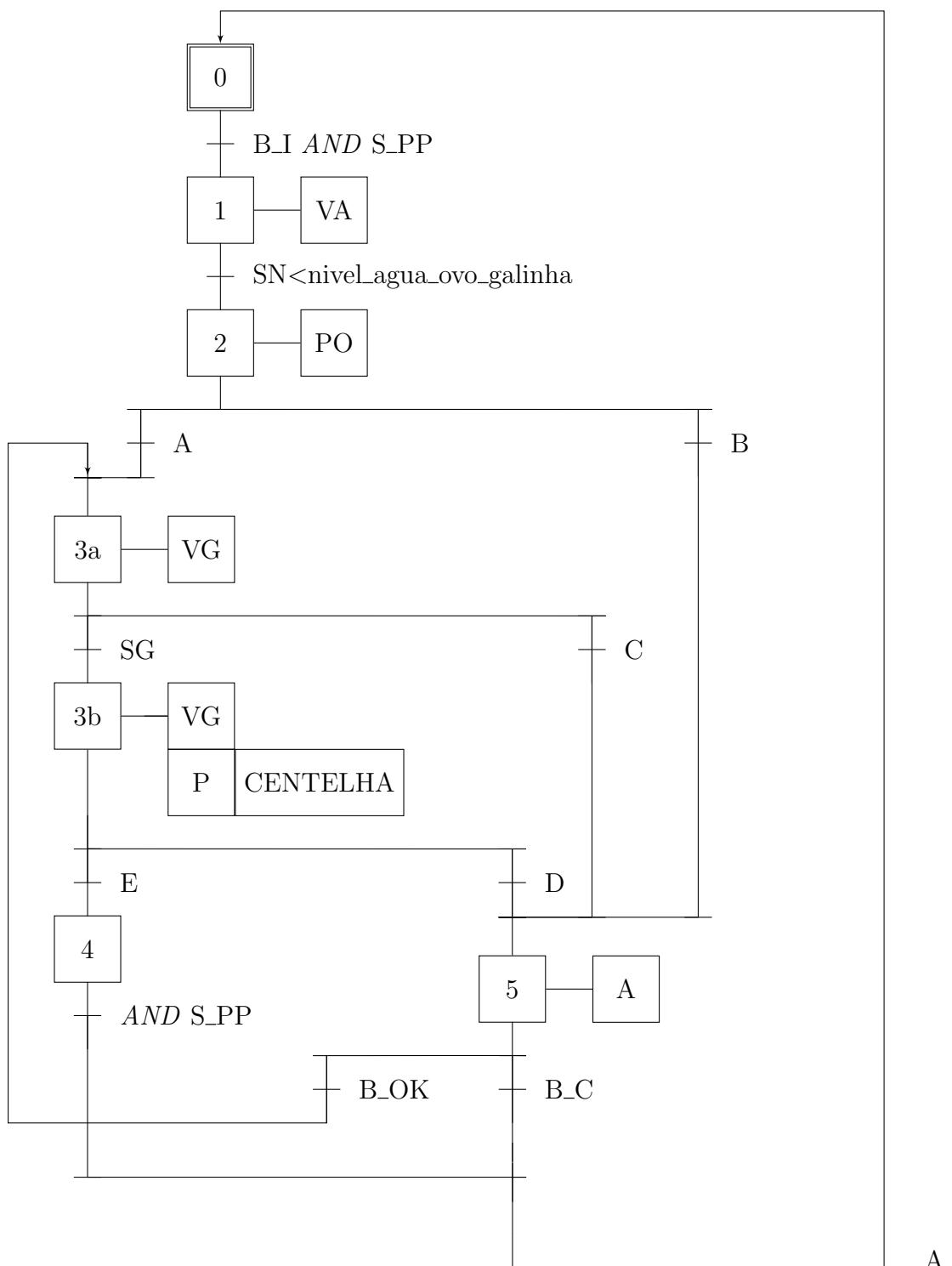


Figura 7.3: Sistema para cozinar ovo com sensores e atuadores.



$= S_N < (\text{nível_ovo_galinha} - \text{volume_ovo_galinha})$

B = NOT A AND 2.T>T#10s

C = NOT SG AND 3a.T>T#4s

D = SG AND 3b.T>T#4s 70

E = NOT SG AND 3b.T>tempo_ovo_galinha

Figura 7.4: Diagrama grafcet nível 2 para cozinhar um ovo.

7.1 Linguagem GrafCet – Aplicações Parte 1

7.2 Linguagem GrafCet – Aplicações Parte 2

Capítulo 8

Instrumentação Industrial

- 8.1 Medição de grandezas mecânicas. Características de instrumentos.
- 8.2 Transmissão de dados, aterramento e blindagem em instrumentação.