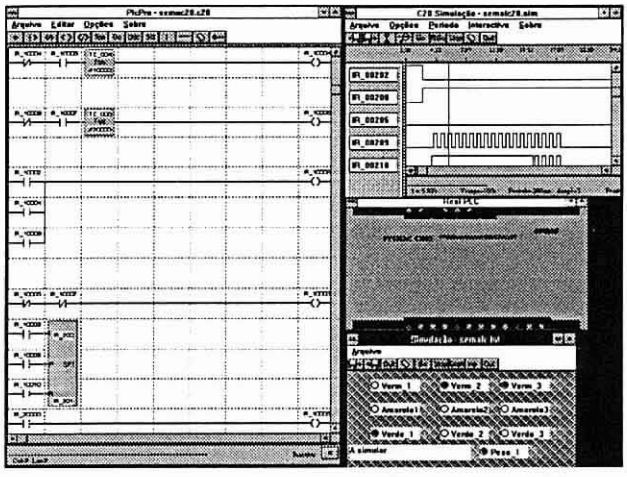
Universidade de Aveiro

Departamento de Electrónica e Telecomunicações



Virtualização de autómatos programáveis

Jorge Augusto Fernandes Ferreira

Lic. em Eng. Electrónica e Telecomunicações

pela

UNIVERSIDADE DE AVEIRO

Abril 1994

Resumo

Os autómatos programáveis (vulgarmente denominados PLCs, do inglês "Programmable Logic Controllers") são bastante usados na indústria. Isto implica que um maior número de pessoas deve obter treino suficiente para desenvolver, manter e melhorar aplicações envolvendo PLCs, desde o simples comando de um motor até sofisticados controladores de processos. Neste trabalho foi desenvolvido um sistema (VPC) para o ensino da programação de PLCs, sem a necessidade de laboratórios de custo elevado, e para testar programas antes de usá-los em condições reais.

A ideia básica é a virtualização de um PLC. Isto significa que o PLC é construído por *software*, por exemplo num computador pessoal, de tal modo que as entradas são introduzidas pelo teclado ou pelo rato, e as saídas são visualizadas no monitor.

O sistema VPC contém três blocos principais: o Editor de programação, o PLC virtual e o Simulador. O Editor suporta a conhecida Linguagem de Contactos (*Ladder*) e fornece as ferramentas necessárias para desenvolver programas para PLCs, utilizando uma interface *point-and-click* baseada em menus e ícones. O PLC virtual e o Simulador implementam as operações comuns de um PLC real, tais como contactos abertos e fechados, saídas, temporizadores, contadores e registos de deslocamento. Ligado ao Simulador existe um editor de estímulos para definir as entradas do PLC virtual.

O PLC virtual pode executar programas em três modos diferentes: a simulação rápida, onde os sinais aparecem sob a forma de diagramas temporais no editor de estímulos, o qual fornece também informação temporal; a simulação em tempo real, onde as entradas e as saídas são visualizadas sobre uma imagem do PLC real, cujo comportamento está a ser emulado, através dos usuais díodos emissores de luz; e a simulação interactiva, também uma simulação em tempo real, onde as entradas são definidas durante o processo de simulação e os resultados são visualizáveis numa janela específica.

Presentemente, o bloco PLC virtual, do sistema descrito, suporta um autómato específico, mas o uso de técnicas de programação por objectos permite uma fácil modificação, de modo a poder emular outros PLCs comerciais. O uso destas técnicas permite também a fácil adição de instruções *Ladder* mais avançadas. O sistema VPC corre como uma aplicação para o ambiente Microsoft Windows.

Palavras Chave: Autómatos Programáveis, Linguagem de Contactos, Diagramas de Escada, PLC Virtual, Virtualização, Programação Orientada por Objectos, Programação em Ambiente Windows, Interfaces Gráficas, Simulação Orientada por Objectos, Simulação em Tempo Real.

Abstract

Programmable Logic Controllers (PLC) are widely used in industry. This implies that more and more people must be trained in order to obtain enough knowledge to develop, maintain and upgrade applications involving PLCs, from simple motor commands to sophisticated process controllers. This project presents a software system (VPC) to teach PLC programming, without the need of costly practical class rooms, and to test programs before using them in real conditions.

The basic idea is the virtualization of the PLC. This means that the PLC is software constructed, for instance inside a personal computer, in such a way that the inputs are obtained with the keyboard or the mouse and the outputs are visualized on the screen.

The VPC system has three main blocks: the Editor, the Virtual PLC, and the Simulator. The Editor, with menus, icons and a point-and-click interface, supports the well known relay ladder programming language and supplies the tools needed to develop a PLC program. The Virtual PLC and the Simulator implement the common operations of a real PLC, such as open and closed contacts, outputs, timers, counters and shift registers. Linked to the Simulator there is a stimuli editor to input the conditions into the Virtual PLC.

The Virtual PLC can simulate programs in three different modes: the fast simulation, where the signals appear as a diagram of logic states inside the stimuli editor, which also provides timing information; the real time simulation, where the inputs and outputs are impressed over an image of the real PLC, whose behaviour is being simulated, with the usual light emitting diodes; and the interactive simulation, also a real time simulation, where the inputs are defined by the user during the simulation process and the results are visualized in a specific window.

Currently, the virtual PLC block of the described system supports a specific PLC, but the use of object oriented techniques makes it easy to change the system, in order to provide support for other commercially available PLCs. The use of these techniques also allows an easy upgrade in order to integrate more advanced ladder instructions, if desired. The VPC system runs as a Microsoft Windows application.

Keywords: Programmable Logic Controllers, Ladder Diagrams, Virtual PLC, Virtualization, Object Oriented Programming, Programming in Windows, Graphic Interfaces, Object Oriented Simulation, Real Time Simulation.

Índice

Capítulo 1
Introdução

1.1. Enquadramento

A ideia da criação dos Autómatos Industriais (PLCs), surgiu na década de 60, devido à necessidade de reduzir o custo inerente às frequentes alterações de sistemas industriais baseados em relés. O primeiro PLC foi instalado em 1969, e provou ser um valioso instrumento na automação industrial.

A utilização deste tipo de equipamento tornou-se, a partir de então, num contributo fundamental para a modernização e para o aumento de produtividade no meio industrial, principalmente nas indústrias de produção [1]. De facto, cada vez mais empresas recorrem a este tipo de equipamento para controlar os seus processos de fabrico, que podem ir desde um pequeno controlo de motores até ao controlo de robôs, incluindo sistemas integrados para processos contínuos. A introdução dos microprocessadores veio permitir o desenvolvimento de PLCs cada vez mais potentes, e acrescentar o número e o desempenho das facilidades fornecidas por estes, nomeadamente controlo analógico, comando numérico, comunicações, entre outros.

Pelo factos apontados acima, a necessidade de um maior número de programadores aumentou, bem como a necessidade de mais e melhor formação, devido à complexidade crescente das aplicações. Assim, a aprendizagem rápida dos conceitos básicos deste tipo de computadores industriais torna-se fundamental. Contudo, a criação de laboratórios para este efeito torna-se onerosa, dado o grande número de PLCs envolvidos.

Por outro lado, é do conhecimento dos programadores de PLCs, o quão difícil é testar um determinado programa em todas as situações. A extrapolação dos resultados de entradas incorrectas devidas a avarias em sensores e/ou interruptores torna-se bastante complexa. Uma possível falha no controlo pode levar sistemas estáveis a comportarem-se de forma imprevisível, com todas as desvantagens daí inerentes para a segurança do processo a controlar, e das pessoas eventualmente envolvidas. A possibilidade de teste de um determinado programa, prevendo um grande conjunto de situações, de um modo rápido e sem necessidade de montar o PLC, é obviamente uma vantagem importante, que assegura um desenvolvimento mais seguro e expedito de aplicações. De facto, esta possibilidade de teste do programa, para vários comportamentos das entradas, pode permitir a detecção de erros de concepção que, de outra forma, só seriam detectados demasiado tarde, possivelmente quando o PLC já estivesse inserido no sistema a controlar.

1.2. Objectivos

As razões apresentadas no ponto anterior foram determinantes na definição dos objectivos deste trabalho, os quais se expressam genericamente no desenvolvimento de um modelo de virtualização para um PLC. Para exemplificar esse modelo deverá ser desenvolvida uma aplicação, a executar em computador pessoal, que emulará o comportamento de um PLC real específico.

A ideia de base para o modelo prevê a construção do PLC por *software*, de forma a que as entradas são obtidas com o rato e/ou teclado e as saídas são visualizadas no monitor (figura 1). O modelo deverá definir a implementação dos componentes principais de um PLC real nomeadamente a memória, o CPU e as unidades de entrada/saída.

A interacção com o utilizador deverá ser simultaneamente amigável e de rápida aprendizagem tendo em conta os propósitos a que a aplicação final se destina, que é o ensino de programação de PLCs e o teste de soluções para problemas de controlo. Para isso, a interface fornecida pelo Microsoft Windows deverá ser a base para a interface gráfica do PLC virtual. A interacção deverá ser feita sobre janelas e privilegiando a utilização do rato, menus e ícones. Deverão então ser construídos editores gráficos e visualizadores que permitam, de uma forma simples e rápida, comunicar com o PLC virtual, desenvolver e armazenar programas e simular a execução destes.

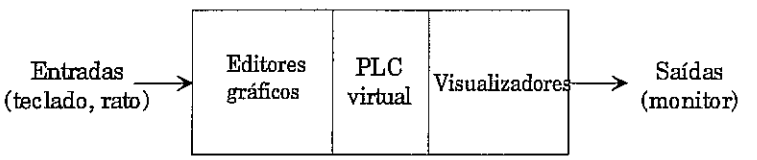


Fig. 1 - Comunicação com o PLC virtual

Para a programação do PLC virtual deverá ser utilizada a linguagem de contactos (*Ladder*), por ser uma linguagem essencialmente gráfica e aceite pela grande maioria dos fabricantes e programadores de PLCs.

Para o desenvolvimento do *software* do sistema propõe-se o uso duma metodologia baseada na programação por objectos. Os aspectos essenciais envolvidos neste trabalho, que estão relacionados com interfaces gráficas e questões de simulação, verão facilitada a sua implementação devido às vantagens das linguagens de programação por objectos, nomeadamente no desenvolvimento de programas de simulação e na construção de interfaces com o utilizador, bem como a facilidade de reutilização de código que estas linguagens proporcionam.

1.3. Estrutura da tese

Esta tese é composta por 7 capítulos e um apêndice. O primeiro capítulo pretende enquadrar o trabalho e traçar, na generalidade, os objectivos a atingir.

Os capítulos 2 e 3 são capítulos introdutórios, que servem para familiarizar o leitor nas áreas chave que este trabalho envolve. Assim, no segundo capítulo é feita uma abordagem à arquitectura dos autómatos programáveis e às suas linguagens de programação. No terceiro capítulo são introduzidas, a partir de alguns exemplos, as ferramentas utilizadas neste trabalho.

Os capítulos 4, 5, 6 discutem o sistema implementado. Assim no quarto capítulo discute-se a interface gráfica para o PLC virtual, e a forma como, genericamente, esta foi implementada. No quinto capítulo discute-se a implementação dos diversos editores gráficos criados para o PLC virtual. No capítulo 6 apresenta-se o nosso modelo para a virtualização de autómatos, e discutem-se os vários modos de simulação de programas pelo PLC virtual.

No capítulo 7 discutem-se algumas vantagens e desvantagens dos modos de simulação implementados, tecem-se algumas considerações sobre possíveis *upgrades* do sistema PLC virtual, e apresentam-se as conclusões e algumas propostas para trabalhos futuros.

Em apêndice (Apêndice A) é apresentado o manual de utilização do sistema PLC virtual.

Nota: sendo este trabalho essencialmente de implementação, decidimos apresentar ao longo do texto algum código de classes em C++ desenvolvidas para o sistema. Queremos com isto fornecer um melhor meio de consulta para possíveis continuadores deste trabalho.

Capítulo 2
Autómatos programáveis

Os autómatos programáveis (vulgarmente denominados PLCs do inglês "Programmable Logic Controllers") são basicamente unidades de *hardware*, com um *CPU* e memória, e que são geralmente utilizados em ambiente industrial para controlo de máquinas e processos.

2.1. História e aplicações industriais

Na década de 60, a necessidade de reduzir os custos devidos às mudanças frequentes nos sistemas de controlo industriais baseados em relés levou ao desenvolvimento dos conceitos associados com os autómatos industriais.

Era suposto que os PLCs iriam diminuir o tempo associado à produção e facilitar, no futuro, possíveis modificações nos sistemas de controlo. Os primeiros autómatos industriais foram instalados no final da década de 60 e logo começaram a provar ser um valioso melhoramento em relação aos painéis lógicos de relés. E isto, porque eram de fácil instalação e programação (ou reprogramação), porque ocupavam menos espaço e porque eram mais seguros do que os sistemas baseados em relés.

Houve dois factores essenciais no desenho do autómato industrial que o levaram a ter o sucesso que indiscutivelmente hoje em dia lhe é reconhecido.

Primeiro, foram usados componentes muito seguros, e os circuitos electrónicos e módulos foram desenvolvidos tendo em conta o ambiente industrial, ou seja foram construídos para resistir ao ruído eléctrico, humidade, óleo, e elevadas temperaturas.

O segundo factor tem a ver com a linguagem de programação inicialmente utilizada, e que era a linguagem de diagramas de escada (do inglês *ladder*), também conhecida por linguagem de contactos, a qual se baseia na lógica de relés. Os sistemas de computadores anteriormente utilizados falharam, porque os técnicos e engenheiros industriais não estavam devidamente treinados nas linguagens de programação usuais de computadores. Contudo, muitos estavam treinados no desenvolvimento de sistemas de controlo baseados em relés, e a programação numa linguagem baseada em circuitos de relés foi rapidamente apreendida.

Na década de 70, os microprocessadores foram introduzidos nos autómatos, e a capacidade e o desempenho aumentaram e melhoraram, ficando assim aptos a realizar tarefas cada vez mais sofisticadas.

Nos finais dos anos 70, as melhorias nos componentes das comunicações e nos circuitos electrónicos permitiram colocar os autómatos a centenas de metros do equipamento que

controlavam, tornando possível a troca de dados entre autómatos de modo a rentabilizar e tornar mais eficiente o controlo de máquinas e processos. O fabrico de módulos de entradas e saídas com conversores analógico/digitais permitiram aos autómatos entrar no área do controlo analógico.

Podemos, hoje em dia, encontrar autómatos nos mais variados sectores industriais. Por exemplo, na indústria metalúrgica, onde a fiabilidade e segurança são factores essenciais, os autómatos são utilizados desde os problemas de análise de gás ao controlo de qualidade. Na indústria mecânica e automóvel, um dos sectores industriais que mais utilizam os autómatos, estes podem ser usados, por exemplo, para controlo de linhas de produção e montagem, controlo de máquinas e robôs [2], [3]. No transporte e empacotamento de materiais os autómatos podem ser utilizados no controlo de gruas e elevadores mecânicos. A utilização de autómatos inclui também as indústrias químicas e petroquímicas, agricultura e indústrias de alimentos, indústria textil, indústria do vidro e plástico, entre muitas outras.

2.2. Arquitectura dos autómatos programáveis

Independentemente do tamanho, e da complexidade, praticamente todos os autómatos programáveis partilham dos mesmos componentes básicos (figura 2) e das mesmas características funcionais [4], [5].

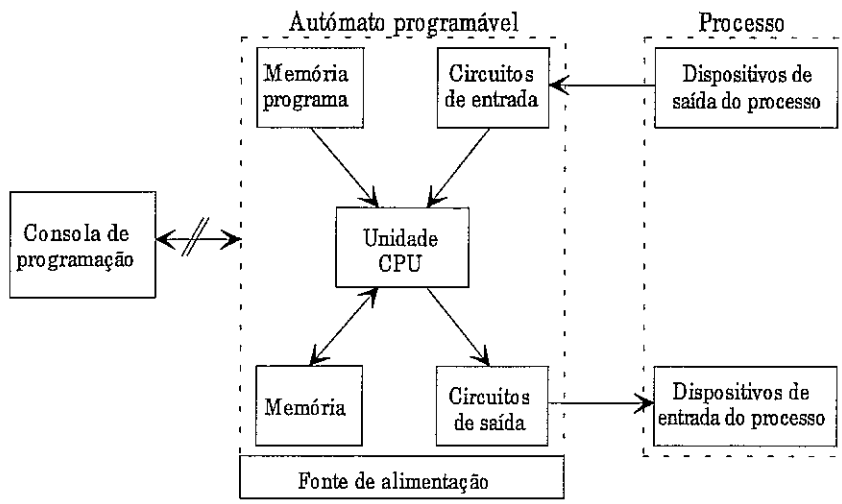


Fig. 2 - Arquitectura de um autómato programável

Um autómato é composto por uma Unidade Central de Processamento (CPU) que executa ciclicamente o programa armazenado na memória de programa. No início de cada ciclo de execução as entradas, que podem provir de um qualquer processo, são analisadas. No final de cada ciclo as saídas são actuadas, de acordo com as entradas e o programa de controlo, de modo a comandar o equipamento ou máquinas envolvidas no processo. Na execução do programa, o CPU utiliza a memória do autómato para armazenamento e transferência de dados. A utilização da memória para armazenamento do programa, separada da outra memória, tem a ver com a necessidade de armazenar os programas no PLC, mesmo com a alimentação desligada.

A consola de programação é geralmente utilizada para o desenvolvimento de programas e para a tranferência dos programas para o autómato. Uma vez inserido o programa no autómato, este é autosuficiente não necessitando da consola quando em execução.

2.2.1. CPU

A Unidade Central de Processamento é responsável pela execução do programa, controlando todas as operações dentro do autómato, através de instruções armazenadas na memória de programa. Um barramento de dados transporta a informação da memória e do sistema de entradas saídas para o CPU e vice versa. Na maioria dos autómatos (principalmente os mais modernos) o CPU é baseado em um ou mais microprocessadores e outros circuitos que permitem realizar as funções de controlo e cálculo necessárias à execução de programas.

2.2.2. Sistema de entradas e saídas

O sistema de entradas/saídas fornece a ligação física entre o CPU e o processo a controlar. O autómato, através de sensores apropriados, pode medir quantidades físicas como velocidade, temperatura, pressão, corrente, etc.. Baseando-se nos valores medidos, e no programa de controlo, o CPU controla as saídas que poderão actuar em dispositivos como, por exemplo, válvulas, motores, alarmes.

O sistema de entradas/saídas é um dos componentes mais importantes num autómato pois estas necessitam de interagir directamente com equipamento industrial e podem residir em zonas de elevado ruído eléctrico. De facto, uma das grandes inovações dos autómatos é a possibilidade de ligação directa aos sensores e actuadores sem haver necessidade de circuitos de adaptação. Para isso as entradas e saídas do autómatos possuem isolamento galvânico (normalmente óptico), o que lhes dá uma melhor fiabilidade e segurança na comunicação com sensores e actuadores. De facto, o isolamento das entradas/saídas é absolutamente necessário por questões de ruído e de modo a compatibilizar os diferentes níveis de tensão e potência existentes entre o autómato e os processos a controlar.

2.2.3. Memória

A memória é usada para armazenar o programa de controlo (memória de programa) e possibilitar o armazenamento e a transferência de dados. Geralmente os autómatos utilizam memória do tipo RAM, EPROM ou EEPROM. Na maioria dos casos a memória do tipo RAM é utilizada nas fases de desenvolvimento e teste dos programas, enquanto que as memórias do tipo EPROM e EEPROM são utilizadas para o armazenamento de programas em código executável e também para armazenamento de configurações do sistema. No entando, hoje em dia, a tendência é para a utilização de memória RAM, devido ao seu baixo consumo, juntamente com baterias que permitem manter o conteúdo da memória mesmo com o autómato desligado.

A capacidade de memória de cada autómato tem em conta as potencialidades de cada um e é geralmente medida em termos do número máximo de instruções de um programa, ou em termos da capacidade de memória em *bytes*. Autómatos pequenos têm geralmente um tamanho de memória fixo, enquanto autómatos maiores permitem a utilização de módulos para expansão da memória.

2.2.4. Variações na arquitectura

A necessidade crescente de autómatos para vários tipos de aplicações, umas mais complexas que outras, levou os construtores a desenvolver várias famílias de autómatos baseadas em processadores com vários níveis de desempenho, permitindo ao utilizador a escolha do tipo de autómato a utilizar de acordo com as necessidades do programa a executar. Por estas razões, surgiram arquitecturas para os autómatos que permitem a configuração de sistemas modulares. O construtor fornece, nestes casos, um mesmo autómato-base ao qual são adicionáveis módulos consoante os requisitos da aplicação. Podemos citar módulos de entradas/saídas digitais, entradas/saídas analógicas, módulos específicos para controlo do tipo PID (proporcional, integral, diferencial), módulos para comunicações, módulos de memória adicional, módulos com contadores rápidos, entre outros.

2.3. Linguagens de programação para autómatos

A programação dos autómatos é feita usando ferramentas de programação, que podem ser consolas fornecidas pelo construtor, ou *software* geralmente executado a partir de um computador pessoal (PC).

Embora tenha surgido há pouco tempo um *standard* (*IEC 1131 Standard - Part 3*, 1993) que define um grupo de linguagens para a programação de PLCs, não nos foi possível usar esse documento em virtude de, embora encomendado, ainda não estar disponível. Assim, as linguagens serão apresentadas com base em descrições de outros autores.

As linguagens de programação dos autómatos podem ser agrupadas em duas grandes categorias [5]: Linguagens Gráficas e Linguagens Literais. As linguagens gráficas, que foram as primeiras a ser utilizadas nos autómatos, fornecem um conjunto de símbolos gráficos que são ligados entre si de forma a constituírem um programa. Por linguagens literais entende-se um conjunto de linguagens cujas instruções são escritas na forma de expressões literais usando partes em texto e palavras reservadas dessas linguagens.

Nas linguagens gráficas podemos destacar a linguagem de diagramas de contactos e a linguagem *GRAFCET*. Das linguagens literais destacam-se as linguagens booleanas, linguagens de mnemónicas e linguagens de processamento de informação do tipo das usadas em programação de computadores.

2.3.1. Linguagem *ladder*

A partir dos diagramas de relés, os construtores americanos derivaram uma linguagem para programação de autómatos industriais, e que tem a grande vantagem de ser parecida com a lógica tradicional de contactos. A maioria dos fabricantes de autómatos fornece, há já bastante tempo, a linguagem *ladder* para a programação dos seus autómatos. Esta linguagem vai ser analisada com maior detalhe pois é a linguagem utilizada neste trabalho, pelas razões que explicitaremos mais adiante.

A linguagem *ladder* consiste numa lista de instruções simbólicas que, quando interligadas entre si, de uma determinada forma, constituem um programa para autómatos. Esta linguagem é composta, segundo HUGHES [4], por seis categorias de instruções que incluem: instruções do tipo relé, temporização/contagem, manipulação de dados, aritméticas, transferência de dados e controlo de programa.

Um programa escrito em linguagem *ladder* consiste em N degraus, em que cada degrau pode representar graficamente uma equação booleana. A principal função destes degraus é a de permitirem controlar saídas a partir de condições de entrada. Tanto as entradas como as saídas podem ser físicas ou pontos internos do autómato (posições de memória usadas para armazenar informação com um formato do tipo *bit*).

A figura 3 mostra a estrutura básica de um degrau. Neste caso, a saída só será actuada quando existir continuidade lógica, isto é quando houver pelo menos um caminho fechado desde o início de continuidade lógica até à saída.

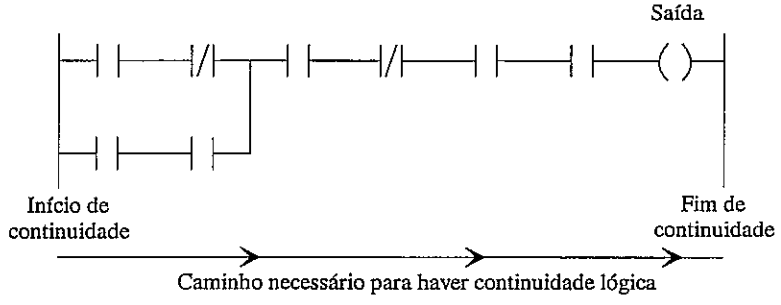


Fig. 3 - Degrau num diagrama de escada

As saídas (também apelidadas de bobinas) e os contactos, são os símbolos básicos da lista de instruções da linguagem *ladder*. Os contactos, programados ao longo de um determinado degrau, representam condições que depois de avaliadas determinam o controlo da saída.

A programação dos contactos e das saídas, consiste na atribuição de endereços que identificam o que está a ser avaliado e o que está a ser controlado. Cada endereço referencia a localização de um ponto interno da memória, ou identifica a saída ou a entrada. Um contacto, independentemente de representar uma entrada ou uma saída, ou um ponto interno, pode ser utilizado em qualquer parte do programa, sempre que aquela condição necessite de ser avaliada.

A organização dos contactos nos degraus depende do controlo lógico desejado. Os contactos podem ser colocados em série, paralelo ou série/paralelo, dependendo do controlo necessário para uma dada saída.

2.3.1.1. Instruções do tipo relé

Estas instruções permitem examinar o estado (*ON/OFF*) de um ponto interno ou entrada, e controlar o estado de um ponto interno ou de uma saída.

Contacto normalmente aberto —| |—

O endereço referenciado pode ser uma entrada, um ponto interno ou uma saída. Se o estado for *ON* quando o contacto está a ser examinado então este fecha-se e assegura a continuidade lógica. Se o estado for *OFF* então sucede o contrário e o contacto abre-se quebrando a continuidade lógica.

Contacto normalmente fechado —|/|—

O princípio de funcionamento é idêntico ao anterior, mas ao contrário. Quando o estado for *OFF* existe continuidade lógica e quando o estado for *ON* não existe continuidade lógica.

Início de ramificação |—

Inicia cada um dos ramos paralelos. É a primeira instrução quando se pretende realizar a função lógica *OR*.

Fim de ramificação —|

Termina um conjunto de ramos paralelos, isto é, termina o caminho de continuidade para um circuito *OR*.

Saída normalmente aberta —( )—

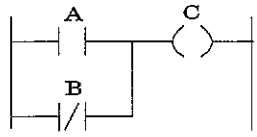
Usada para contrólar uma saída ou um ponto interno. Coloca no estado *ON*, a saída ou o ponto interno, quando existe continuidade lógica, e vice versa.

Saída normalmente fechada —(/)—

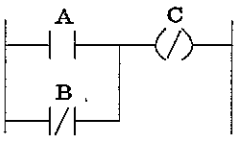
O seu comportamento é o inverso da saída normalmente aberta.

Exemplos:

O degrau que corresponde à descrição em linguagem *ladder* da equação booleana $A+\overline{B}=C$, é o seguinte:

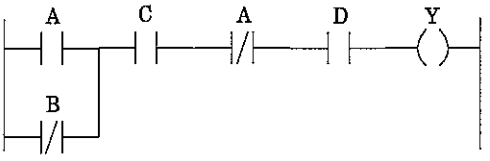


Para a equação booleana $\overline{A+\overline{B}}=C$, o degrau será:

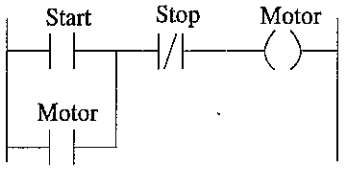


Como se pode verificar com estes dois exemplos, a realização de um *OR* lógico corresponde à ligação em paralelo de dois símbolos, enquanto que para a realização de uma negação lógica basta a utilização de contactos ou saídas normalmente fechadas.

A seguir, apresenta-se um exemplo para se verificar que um *AND* lógico se realiza através da ligação em série de contactos. Para a equação lógica $(A+\overline{B})\cdot C\cdot\overline{A}\cdot D=Y$, o degrau correspondente será:



Vejamos agora um exemplo bastante simples, mas concreto: o arranque e a paragem de um motor com as tradicionais botoneiras *Start* e *Stop*. Como é bem conhecido, para comandar um motor a partir de botoneiras é necessário fazer a chamada auto-alimentação. É essa a função do relé Motor, que garante que o motor continue a trabalhar quando se deixa de premir a botoneira *Start*. O degrau para comandar desta forma o motor seria:



Quando o botão de *Start* é premido, e o botão de *Stop* não está activo, existe continuidade lógica, o motor arranca e o relé Motor é accionado. Mesmo que o botão de *Start* seja agora levantado, a continuidade lógica é assegurada pelo contacto Motor.

Para parar o motor basta premir o botão de *Stop*, quebrando a continuidade lógica, e neste caso também o relé Motor deixa de estar accionado. Mesmo que o botão de *Stop* seja agora levantado, continua a não haver continuidade lógica pois quer o *Start* quer o Motor estão abertos.

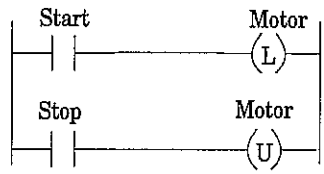
Saída *latched* —(L)—

Esta saída é colocada a ON quando existir pelo menos um caminho com continuidade lógica. No entanto, esta continuará ON mesmo que deixe de existir continuidade lógica. A saída só será colocada inactiva quando for executada uma instrução de *unlatched* para o mesmo endereço.

Saída *unlatched* —(U)—

É programada para colocar OFF uma determinada saída *latched.* É a única forma de colocar OFF uma saída *latched.*

Com estas duas instruções o exemplo do motor apresentado anteriormente poderia ser realizado pelos dois degraus apresentados a seguir:



2.3.1.2. Instruções de temporização e contagem

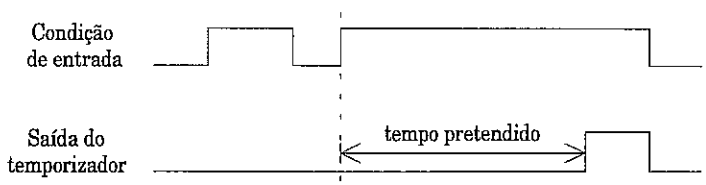
São instruções de saída com funções idênticas às fornecidas pelos temporizadores e contadores construídos mecânica ou electronicamente. São geralmente utilizadas para activar ou desactivar um dispositivo ao fim de determinado tempo ou contagem.

O seu princípio de funcionamento é idêntico pois ambos podem ser considerados contadores. Um temporizador conta um número de intervalos de tempo fixos, necessário para atingir a duração pretendida, enquanto que um contador regista o número de ocorrências de um determinado evento.

As instruções de temporização e contagem necessitam de dois registos: um registo para armazenar o número de contagens já efectuadas e outro registo para armazenar o valor inicial.

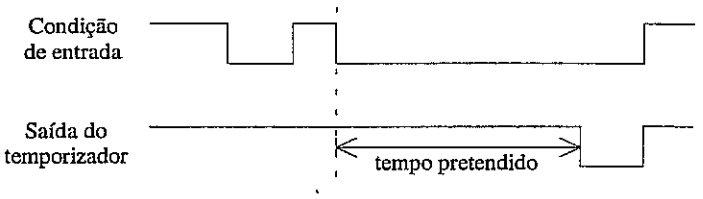
Atraso à operação TON

Usada para providenciar atraso numa determinada acção, ou para medir a duração da ocorrência de um evento. A temporização é iniciada quando a condição de entrada é verdadeira, sendo o temporizador inicializado quando a condição de entrada deixar de ser verdadeira. No diagrama temporal apresentado a seguir podem-se analisar os sinais da condição de entrada e o sinal de saída do temporizador.



Atraso à desoperação TOFF

Usada para atrasar a desactivação de uma determinada saída ou ponto interno. Se a continuidade lógica for quebrada por algum motivo, o temporizador inicia a contagem do tempo respectivo ao fim do qual coloca a sua saída inactiva, como se pode analisar na figura seguinte:



Estes dois tipos de temporizadores são os mais utilizados pela maioria dos fabricantes de autómatos para as suas implementações da linguagem *ladder*.

Contador ascendente CTU

A instrução de CTU incrementa de uma unidade o seu valor acumulado de cada vez que ocorre um evento de contagem. Ao contrário das instruções de temporização, o contador, na maioria das implementações, continua a contar eventos mesmo após o seu valor de contagem ser atingido.

Contador descendente *CTD*

O princípio de funcionamento é idêntico ao contador ascendente, só que o processo de contagem é descendente desde o valor pré-programado até zero.

Também é normal haver outras implementações de contadores nomeadamente o contador ascendente/descendente, bem como a introdução de uma entrada para inicialização (*reset*) em cada um dos contadores atrás citados. Quando a entrada de *reset* está inactiva o contador tem um comportamento normal, e uma activação da entrada de *reset* inibe imediatamente a contagem e inicializa o contador com o seu valor pré-programado.

2.3.1.3. Instruções de manipulação e transferência de dados

A evolução da linguagem *ladder* veio permitir a inserção de módulos na linguagem. De facto a maior parte dos fabricantes não utiliza somente instruções *ladder* com uma entrada e uma saída. Nesta abordagem, considera-se como um módulo as instruções que contêm mais do que uma entrada e/ou saída, ou em que a acção da instrução envolve mais do que um *bit*. A introdução deste tipo de instruções permite a utilização desta linguagem para a resolução de problemas de maior complexidade.

Como módulos encontram-se instruções para transferências de palavras de uns registos para outros, conversão de formatos das palavras (por exemplo de formato binário para formato BCD e vice versa), instruções de deslocamento (*shift registers*), entre outros. Em quaisquer destas instruções são especificados os endereços da palavra fonte e da palavra destino.

Outro tipo de instruções são as de comparação de dados, através das quais é possível comparar palavras armazenadas em dois endereços. A sua saída é geralmente em *bit*, o que permite elaborar estruturas de decisão com base na comparação de dados com mais de um *bit*.

2.3.1.4. Instruções aritméticas

Estas instruções incluem as quatro operações aritméticas básicas: adição, subtracção, multiplicação e divisão. Existem, no entanto, autómatos que fornecem outro tipo de operações aritméticas como, por exemplo, a raiz quadrada, complemento, incremento/decremento.

2.3.1.5. Instruções de controlo de programa

Este tipo de instruções é usado a fim de alterar a sequência de execução do programa. As instruções de controlo de programa tornam assim possível a execução de partes de programa de uma forma condicional, e além disso, permitem a utilização do conceito de subrotina na linguagem *ladder*. A subrotina toma geralmente a forma de um módulo definido pelo utilizador.

As instruções mais representativas são apresentadas a seguir:

JMP ***label***- a próxima instrução a executar é a instrução cujo endereço é *label*.

LABEL - permite especificar um endereço na forma textual.

CALL - ***label*** chama uma subrotina cujo código está localizado no endereço especificado em *label*. A instrução a executar após a execução da subrotina, é a instrução localizada a seguir à instrução de CALL.

RET - instrução para marcar o fim de uma subrotina.

END - instrução que marca o fim do programa.

As instruções de chamada a subrotinas permitem uma modularização dos programas, pois possibilitam a execução de módulos pré-definidos pelo programador, rentabilizando a memória existente para a criação de programas.

Muito embora o leque de instruções *ladder* implementadas pelos diversos fabricantes seja muito variado, dependendo na maioria dos casos da gama do autómato, abordaram-se neste ponto os tipos de instruções mais geralmente utilizadas em aplicações de pequena complexidade.

2.3.2. O *GRAFCET*

Foi originalmente uma metodologia aplicada na representação gráfica de algoritmos usados em sistemas de controlo. Devido à utilização de símbolos gráficos para a representação desses algoritmos e à boa legibilidade na análise funcional dessas representações, alguns construtores transformaram o *grafcet* numa linguagem de programação.

Um diagrama em *grafcet* engloba, basicamente, três entidades distintas que são as etapas, as condições de transição e as tarefas (figura 4).

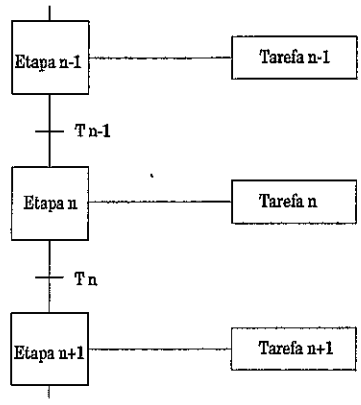


Fig. 4 - Etapas, transições e tarefas no *grafcet*

Cada etapa, definida como uma situação num sistema de controlo onde as entradas e/ou saídas não variam, tem associada uma acção ou tarefa (conjunto de acções) específica, que só é executada quando a etapa correspondente se encontra activa.

A operação sequencial do processo resulta da progressão entre etapas. Contudo a passagem de uma etapa para outra é condicionada pela condição de transição.

Para que exista uma transição de uma etapa para outra é necessário que a etapa anterior esteja activa e a condição de transição entre as duas etapas seja verdadeira. Por exemplo (figura 4), para que a etapa **n** fique activa é necessário que a etapa **n-1** esteja activa e a condição de transição **T n-1** seja verdadeira. A transição completa-se com a inactividade da etapa **n-1**.

Esta linguagem não substitui as outras linguagens, e deve ser vista como uma ferramenta de estruturação de programas. A sua utilização tem vantagens sempre que o problema a resolver é sequencial e dependente do tempo [6].

2.3.3. Linguagens booleanas

As linguagens booleanas permitem transcrições directas de equações booleanas. Esta linguagem foi inicialmente baseada nos operadores booleanos como os operadores *AND*, *OR* ou *NOT*. Por exemplo, poder-se-ia introduzir directamente a equação lógica $(A \cdot B + \overline{C}) \cdot D = Y$ que correponderia ao cálculo do primeiro membro e à atribuição do resultado ao segundo membro.

2.3.4. Linguagens de mnemónicas

As linguagens de mnemónicas usam o formalismo da linguagem *assembler*. Uma linguagem deste tipo corresponde na prática à utilização directa das instruções fornecidas pelo autómato. O maior interesse destas linguagens é a possibilidade do programador tirar vantagens de todas as potencialidades oferecidas pelo autómato, muito embora para aplicações sequenciais tornem a programação mais complicada e confusa. Por outro lado, a escrita do programa é mais penosa, dificultando também a correcção de erros nos programas.

2.3.5. Linguagens de processamento de informação

As linguagens de processamento de informação surgiram nos autómatos por influência dos computadores, e podem ser consideradas como de alto nível. São utilizadas para resolver problemas mais complexos, com destaque para o controlo analógico, controlo PID, controlo de eixos, manipulação de grandes quantidades de dados, entre outros que seriam de difícil resolução utilizando as linguagens mencionadas nos pontos anteriores. De entre estas linguagens podemos citar as linguagens *BASIC*, *C* e *PASCAL*.

2.4. Breve descrição do autómato a emular

Como um dos objectivos deste trabalho é a contrução de um simulador que execute as instruções de um autómato real, torna-se necessário concretizar esse equipamento, de modo a comparar os resultados de ambos os sistemas.

O autómato escolhido foi o modelo C20H da OMRON, por ser um modelo razoalvelmente potente e estar disponível no ambiente de trabalho onde estávamos inseridos.

Segue-se uma apresentação breve do *hardware* do autómato utilizado, deixando-se a descrição das instruções para um momento posterior, contemporâneo da análise da implementação das instruções do PLC virtual.

2.4.1. Configuração do autómato C20H

Na configuração base, o autómato fornece 12 entradas e 8 saídas; no entanto, o número de entradas e saídas físicas pode ir até um máximo de 48 entradas e 32 saídas, utilizando unidades de expansão. O modelo C20H contém uma interface RS-232, que pode ser utilizada para comunicação com um terminal ou um computador. Em termos de linguagem *ladder* a OMRON fornece para este autómato 77 instruções [7].

2.4.2. Configuração da memória

Para facilitar a gestão de dados existem diversas áreas de memória (memória para dados), cuja utilização depende do tipo de instruções a executar. Estas diferentes áreas de memória permitem separar os vários tipos de dados existentes num programa. Outra área de memória disponível é a reservada para programas (memória de programa). A tabela seguinte apresenta alguns detalhes das várias áreas de memória existentes para o C20H.

Área de memória	Designação	Gama de variação	Função
Internal Relay	*IR*	Palavras: 000 --- 246 *Bits*: 00000 --- 24615	Usada para controlar entradas ou saídas, contadores, temporizadores, e para armazenar dados.
Special Relay	*SR*	Palavras: 247 --- 255 *Bits*: 24700 --- 25515	Contém os relógios do sistema, *flags*, *bits* de controlo, e informação de estado.
Auxiliary Relay	*AR*	Palavras: 00 --- 27 *Bits*: 0000 --- 2715	Contém *flags* e *bits* para funções especiais. Guarda o estado durante falha de energia.
Data Memory	*DM*	Leitura/Escrita: 0000 --- 0999 Somente Leitura: 1000 - 1999	Usada para armazenamento e manipulação de dados internos.
Holding Relay	*HR*	Palavras: 00 --- 99 *Bits*: 0000 --- 9915	Usada para armazenar dados e para guardar valores de dados quando o PLC é desligado.
Timer/Counter	*TC*	000 --- 511	Usada para definir temporizadores e contadores, e para aceder às *flags* PV e SV.
Link Relay	*LR*	Palavras: 00 --- 63 *Bits*: 0000 --- 6315	Disponíveis para uso como *bits* de trabalho.
Temporary Relay	*TR*	*Bits* 00 --- 07	Usado para armazenar e ler condições de execução.
Program Memory	*UM*	Depende da Unidade de Memória usada.	Contém o programa a ser executado pelo CPU.

Em geral, para aceder a uma determinada posição de memória é utilizada em primeiro lugar a designação, depois o número de palavra e, em seguida, o número do *bit* (se for caso disso).

Área de memória *Internal Relay*

Esta área de memória é acedida ou por *bit* ou por palavra, e está compreendida entre as palavras 000 e 246. Nesta área encontram-se os *bits* ou palavras correspondentes às entradas e às saídas físicas, e ainda os *bits* ou palavras para trabalho. Estes *bits* ou palavras para trabalho são normalmente usados, ao longo do programa, para armazenamento de informação.

Para o autómato C20H a palavra de entrada é IR000, enquanto a palavra de saída é IR002, o que significa que, em termos de *bits*, as entradas estão compreendidas entre IR00000 e IR00011 e as saídas entre IR00200 e IR00207.

Para as unidades de expansão, as palavras de entrada e de saída dependem do número da unidade. Por exemplo, para a primeira unidade de expansão a palavra de entrada é IR010 e a palavra de saída IR012.

Área de memória *Special Relay*

Esta área de memória contém *flags* e *bits* de controlo usados para monitorar as operações do autómato, aceder aos impulsos do relógio e sinalizar erros.

Área de memória *Auxiliary Relay*

A maior parte desta área de memória é dedicada a usos específicos como, por exemplo, *flags*, *bits* de controlo, não podendo ser usada para outros fins. Esta área mantém o conteúdo durante interrupções de energia ou quando o autómato entra no modo de programação.

Área de memória *Data Memory*

Esta área só pode ser acedida por palavra e não por *bit*. É utilizada para a escrita ou leitura de dados, particularmente em instruções de tranferência de dados. Outra parte desta memória é usada para guardar parâmetros do sistema.

Área de memória *Holding Relay*

É usada para armazenar ou manipular vários tipos de dados e pode ser endereçada quer por *bit*, quer por palavra. Esta área mantém o seu conteúdo quando o modo de operação do autómato muda, ou quando a alimentação é interrompida.

Área de memória *Timer/Counter*

É utilizada para a criação de contadores ou temporizadores e contém os *bits* de saída, os valores iniciais e os valores actuais de todos os temporizadores e contadores.

Área de memória *Link Relay*

Esta área é utilizada como uma área de dados comum, para autómatos desta família que permitem ligações entre si.

Área de memória *Temporary Relay*

Tem capacidade para oito *bits* e é usada para permitir certos tipos de ramificações na programação em linguagem *ladder*.

Área de memória *Program Memory*

É usada para armazenar o programa a executar no autómato.

Capítulo 3
Ferramentas utilizadas

A interface gráfica construída para este trabalho é baseada na interface gráfica fornecida pelo Microsoft Windows. Neste capítulo vão ser apresentadas as razões da escolha desse ambiente como interface entre o PLC virtual e o utilizador, bem como as razões que determinaram a utilização duma livraria de classes em C++ (Win++) para a comunicação da aplicação com o Microsoft Windows. Para um melhor entendimento da implementação da interface gráfica são também analisadas algumas das particularidades do Microsoft Windows, da linguagem C++ e das classes do Win++. Serão ainda abordados alguns aspectos sobre o desenvolvimento de aplicações para o Windows, com base na biblioteca de classes do Win++.

3.1. O Microsoft Windows

A escolha do Microsoft Windows para a base da interface gráfica do sistema teve em consideração vários factores, dos quais se destacam os seguintes:

- A decisão da utilização de um computador pessoal (PC) para esta aplicação. Esta plataforma parece ser a mais adequada tendo em conta a proliferação de PCs no mundo industrial e também no mundo do ensino [8], dado o seu baixo custo quando comparado com outro tipo de computadores, como sejam por exemplo as estações de trabalho.
- Parece óbvio que um sistema de janelas seria o mais apropriado para este tipo de aplicação. Como é do conhecimento dos programadores de PLCs, o *software* de apoio à programação fornecido actualmente pela maioria dos fabricantes é, em geral, antiquado no que diz respeito à interface com o utilizador. De facto, o tempo dispendido na aprendizagem de uma nova interface de programação pode ser em alguns casos bastante significativo. Em contrapartida, os sistemas com janelas, sob o ponto de vista de interacção com o utilizador, permitem uma aprendizagem mais rápida de uma nova aplicação. Por estas razões, pode aqui afirmar-se que a interface com o utilizador é fundamental, pois quanto mais intuitiva e universal ela for, menor é o tempo de aprendizagem das suas características.
- Analisando as revistas da especialidade, verifica-se que as aplicações DOS estão a desaparecer, para dar lugar a aplicações que correm sobre o Microsoft Windows, pelo menos quando a interface com o utilizador é fundamental como em editores, folhas de cálculo, gráficos, simuladores. Actualmente, a interface do Windows é conhecida pela maior parte dos utilizadores de computadores pessoais, donde a aprendizagem de uma nova aplicação que corra nesse ambiente fica facilitada, já que a base da sua interface é idêntica à de muitas outras aplicações.

Seguidamente serão revistos globalmente os princípios de funcionamento do Microsoft Windows, e apresentados sucintamente os diferentes componentes de interacção com o utilizador utilizados na interface gráfica do Windows.

O Windows fornece uma interface gráfica (GUI) que facilita a criação de programas interactivos. Esta GUI fornece um conjunto de objectos de interacção com o utilizador, como por exemplo menus, janelas e ícones, permitindo que todos os programas para o Windows possuam a mesma aparência, tornando-os mais fáceis de apreender e usar [9].

A estrutura dos programas para Windows é diferente da estrutura dos programas para o MS-DOS, e é parecida com outras GUI, como por exemplo Apple Macintosh, OS/2 Presentation Manager, Motif e XView, que correm em ambientes que são geridos por eventos (*event driven),* isto é, a estrutura de operação deste tipo de programas é centrada à volta de eventos gerados pelo utilizador (teclado ou rato).

A programação *event driven* é suportada através do sistema de mensagens do Windows. A ocorrência de um evento gera uma mensagem que é colocada numa fila de mensagens. Estas, por sua vez, são enviadas uma a uma para as respectivas aplicações.

As mensagens são importantes para a programação em Windows, já que, de facto, a maior parte do trabalho do programador consiste em decidir quais as mensagens que vai processar e em desenvolver o código para atender a cada mensagem.

Um programa, ao processar uma determinada mensagem, aguarda o aparecimento de outra mensagem. Quando o utilizador necessita de trabalhar com um programa, ele foca a sua atenção na janela utilizando o rato ou o teclado. Esta acção faz com que as mensagens sejam enviadas para a aplicação ou janela que contém o *focus* de mensagens, isto é, a que está activa de momento.

As mensagens fornecem a entrada para o programa, enquanto que a saída do programa é somente a saída gráfica. Esta forma de saída gráfica é diferente da filosofia dos sistemas tradicionais onde a saída é geralmente em texto.

Usando uma filosofia independente dos dispositivos gráficos (GDI), o Windows permite que um programa escreva da mesma forma em qualquer dispositivo, isto é, recorrendo à chamada das mesmas subrotinas. Por exemplo, para escrever numa janela ou imprimir numa impressora, utilizam-se as mesmas funções, mas actuando sobre dispositivos diferentes. Assim sendo, cada dispositivo gráfico possui uma janela de saída gráfica própria, sendo, o Windows responsável pelo *clip* automático para essa janela, isto é, a saída é limitada pelo bordo da janela.

3.1.1. Objectos gráficos de interface

O Windows suporta um conjunto de objectos gráficos de interface, nomeadamente janelas, ícones, menus, caixas de diálogo [10]. A forma como o Windows suporta estes objectos gráficos permite aos programadores utilizá-los nas suas aplicações, sendo mínimo o esforço para a sua criação e manutenção. Cada objecto gráfico tem associado um conjunto de mensagens que permitem a interacção com a aplicação.

O objecto gráfico mais importante é a janela, já que qualquer programa que possibilite interacção com o utilizador terá que ter uma janela, pois esta permite receber mensagens do rato e do teclado e permite uma saída gráfica.

Janelas

Do ponto de vista do utilizador, uma janela permite a visualização de dados, mas também pode identificar uma aplicação. Quando o utilizador inicia uma aplicação, aguarda que surja uma janela, e o fecho da janela termina a aplicação. Para decidir qual a aplicação a usar o utilizador selecciona a janela correspondente.

Do ponto de vista do programador, uma janela serve para organizar objectos gráficos de interacção e para dirigir o fluxo de mensagens do sistema. As entradas são enviadas para a janela que, por sua vez, as envia para a aplicação. As aplicações podem subdividir-se em várias janelas, que são criadas obedecendo ao *template* fornecido pelo Windows.

Ícones

Um ícone é um símbolo que funciona como um memorando para o utilizador. As GUI são construídas considerando que o concreto e visível é de mais fácil compreensão do que o que é abstracto e invisível. Desta forma o Windows permite aos utilizadores não memorizar informação, já que com ícones é possível representar, de uma forma gráfica, comandos, programas ou dados.

Menus

Um menu é uma lista de comandos e de opções de um programa. O Windows define vários tipos de menus, sendo os mais utilizados os menus de sistema, menus em barras, os menus tipo *pop-up* e os menus encadeados.

Barras de deslocamento (*Scroll bars*)

Quando um *scroll bar* é realçado, o utilizador sabe que a área real da janela é maior do que a área visível, e então os *scroll bars* poderão ser usados para visualizar todos os objectos contidos na janela.

Cursores

O cursor é um *bitmap* que desliza no ecrã em resposta a um movimento do rato ou de outro dispositivo apontador. A aparência do cursor pode ser modificada de modo a poder sinalizar o estado ou uma mudança no programa.

Carets

Também é um *bitmap* e serve como um ponteiro para a entrada pelo teclado. A janela que tem o controlo do teclado (janela que tem o *focus*) pode criar um *caret* para notificar o utilizador desse facto. A interface do Windows apenas suporta um *caret* de cada vez, devendo as aplicações

que necessitem do *caret* criar um quando têm o *focus* de entrada, devendo destruí-lo quando perdem o *focus*.

Caixas de diálogo

Estes fornecem uma forma *standard* para entrada de dados. Uma caixa de diálogo é uma janela que suporta outras janelas que, ou fazem o *display* de informação, ou permitem a entrada de dados pelo utilizador. Estas pequenas janelas designam-se por controlos de uma caixa de diálogo.

3.2. Programação orientada por objectos

O presente trabalho utiliza técnicas de programação por objectos, pelo que se segue uma pequena introdução aos conceitos inerentes à utilização destas técnicas.

A programação orientada por objectos pode considerar-se, de uma forma simplista, uma técnica de programação que engloba quatro conceitos distintos: classes, encapsulamento, hereditariedade e polimorfismo [11], [12], [13] e [14].

3.2.1. Classes

Uma classe pode ser considerada uma abstracção dos dados referentes a uma entidade, que pode ser real ou não. Uma classe é definida por um conjunto de dados (campos) e por um conjunto de métodos (procedimentos), que pertencem unicamente à classe e que caracterizam a entidade que está a ser representada.

3.2.2. Encapsulamento

Os dados da classe podem ser privados, fechando, desta forma, a classe para o exterior. Contudo o acesso a esses dados pode ser permitido através de métodos da própria classe. Esses métodos fazem parte da interface da classe com o exterior. A esta característica chama-se encapsulamento ou dados escondidos (*data hiding*), porque o acesso a esses dados é restrito a uma lista de métodos declarados e definidos explicitamente pelo programador. O estado de um objecto está contido nas suas variáveis privadas, visíveis somente pelos métodos do próprio objecto. O encapsulamento reduz a ocorrência de erros, permitindo o acesso a dados exclusivamente através de métodos pré-definidos.

3.2.3. Hereditariedade

O conceito de herança é o que mais distingue a programação orientada por objectos das outras filosofias de programação. As classes, uma vez definidas, podem ser usados para construir hierarquias de classes. Nestas hierarquias, as classes podem herdar características de outras classes

definidas acima na hierarquia. Este conceito de hereditariedade reduz a duplicação de código, e contribui para um desenvolvimento de *software* mais eficiente.

A herança providencia uma forma de definir uma nova classe de objectos, adicionando novos dados ou métodos a classes previamente definidas. Se uma classe, B, é herdeira de uma classe A, todos as características (dados e métodos) desta ficam automaticamente disponíveis na classe B, sem haver necessidade de uma nova definição. A classe B pode por seu lado adicionar novas características para os seus fins específicos. O conceito de herança permite ainda que métodos da classe A possam ser redefinidos na classe B; esta redefinição corresponde à reescrita local, para a classe B, do código adequado ao método redefinido.

Segundo MEYER [14], um dos maiores problemas no desenvolvimento de software reutilizável é a necessidade de ter em conta as características comuns aos grupos de dados abstractos relacionados entre si. Como já foi referido atrás, utilizando o mecanismo da herança podem-se construir hierarquias de classe ligadas por relações de herança. Neste contexto, a ideia é mover a definição de cada característica o mais acima possível na hierarquia de classes, para que possa ser partilhada (sem necessidade de repetir a sua definição) pelo maior número possível de classes de níveis mais abaixo. A herança é assim uma técnica fundamental para a reutilização de *software*.

3.2.4. Polimorfismo

O último conceito, polimorfismo, permite a existência de métodos com o mesmo nome acima e abaixo numa hierarquia de classes, com cada classe a implementar a acção (método) de uma forma apropriada para si própria. O conceito de polimorfismo, e o seu complemento principal, a ligação dinâmica (*dynamic binding*), permitem que as operações e as acções (respostas a mensagens) se adaptem automaticamente ao objecto sobre o qual são aplicadas, o que significa que o sistema, quando em execução, selecciona automaticamente a versão da operação correspondente ao objecto que recebe a mensagem, isto é, a mesma mensagem pode originar respostas diferentes consoante o receptor.

O polimorfismo permite, assim, consistência no envio de mensagens para objectos de classes que têm um antepassado comum.

3.2.5. Metodologia de programação por objectos

A tarefa do programador no desenvolvimento de *software* é a de decidir sobre e definir quais as classes de objectos a utilizar na sua aplicação. As classes devem ser implementadas, tanto quanto possível, como unidades úteis e interessantes por si próprias, independentemente dos sistemas a que pertencem, possibilitando desta forma o uso das mesmas classes noutras aplicações. O desenvolvimento de programas baseia-se em melhoramentos sucessivos dessas classes de objectos. A programação com classes muda a ênfase do desenvolvimento de algoritmos para o desenvolvimento de classes.

Cada classe é uma representação directa de um conceito ou de uma entidade no programa; cada objecto manipulado pelo programa é de uma classe específica que define os seus comportamentos. Por outras palavras, qualquer objecto num programa é de uma classe que define a lista de operações possíveis no objecto.

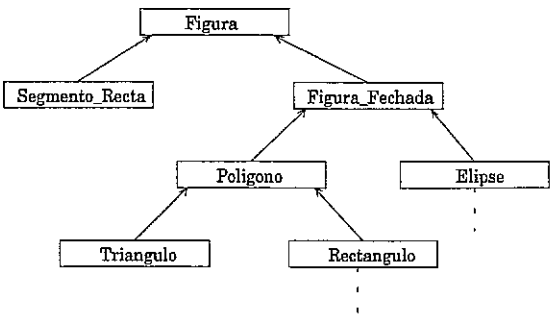
A estruturação do *software* resulta do estabelecimento das relações entre os objectos envolvidos na aplicação, nomeadamente relações de cliente e de descendente. Uma classe é considerada cliente de outra quando faz uso dos seus métodos (uso de objectos do tipo dessa classe), e é considerada descendente de uma ou mais classes quando é desenvolvida como uma extensão ou especialização dessas classes.

3.3. A linguagem C++

O C++ [15], [16], como muitas outras linguagens, surgiu como uma ferramenta para resolver um problema específico. Bjarne Stroustrup, um investigador da Bell Labs, necessitou de escrever alguns programas de simulação. A linguagem Simula 67, a primeira linguagem orientada para objectos seria a linguagem ideal para a escrita do código para esses programas se não fosse a sua baixa velocidade de execução. Stroustrup decidiu então escrever uma nova versão de C, à qual chamou "C com classes". Passados alguns anos essa linguagem desenvolveu-se consideravelmente e o nome foi modificado para C++. O termo "++" sugere que a linguagem C++ é um pouco mais que o C, oferecendo melhoramentos através do suporte de dados abstractos e programação orientada por objectos.

Desde a sua primeira implementação que a linguagem C++ tem sido cada vez mais utilizada, existindo hoje em dia implementações para as mais variadas máquinas. Esta linguagem tem vindo a ser aplicada em muitas áreas da programação, nomeadamente na construção de interfaces com o utilizador e no desenvolvimento de algoritmos genéricos para simulação.

Não é nosso objectivo descrever a linguagem C++, mas sim rever sucintamente alguns conceitos e mecanismos que o C++ utiliza para a implementação da filosofia de programação por objectos. Essa apresentação baseia-se num exemplo, para a hierarquia de classes seguinte.



Como sabemos, muitos conceitos podem ser relacionados com outros conceitos das mais variadas formas. Por exemplo, os conceitos de avião e de carro relacionam-se com os conceitos de veículos e transportes; conceitos de mamífero e pássaro fazem parte de um conceito mais geral de

animais vertebrados. No exemplo proposto, os conceitos de círculo, rectângulo e segmento de recta envolvem o conceito mais geral de figura.

Num programa, a representação de um conceito como um tipo, requer processos para exprimir as relações entre os tipos. O C++ permite especificar hierarquias organizadas de classes, sendo esta a característica chave do suporte para a programação orientada por objectos.

Consideremos um tipo definido, Figura, para uso num sistema gráfico. O sistema tem de suportar círculos, triângulos, rectângulos, elipses e muitas outras figuras. Em primeiro lugar especifica-se uma classe que define as propriedades gerais de todas as figuras, estabelecendo o conceito mais geral deste exemplo.

```
class Figura
{
private:
    ponto centro;
    int cor;
public:
    Figura():centro(0,0);{cor = 0;}
    Figura(ponto cent, int co):centro (cent){ cor = co; }
    ~Figura(){}
    ponto centro_Figura(){return centro;}
    int cor_figura{return cor;}
    void translacao(ponto novo_centro) {centro=novo_centro; desenho();}
    virtual void desenho() = 0;
    virtual void rotacao(int) = 0;
};
```

Podemos definir já a interface de chamada para as funções desenho() e rotacao(), mas ainda não é possível definir a sua implementação. Estas funções são então, declaradas "virtual" e vão ser definidas para cada figura específica. Aplicando os mecanismos de polimorfismo e ligação dinâmica, pode-se querer especificar funções para desenho e de rotação ao nível de Figura, e obrigar a que cada classe terminal, descendente de Figura, forneça as implementações para estas duas funções. Isto é conseguido declarando as funções como funções virtuais puras (= 0). Uma classe virtual é uma classe que contém pelo menos uma função virtual.

Dada este definição da classe Figura, podem escrever-se funções gerais de manipulação de figuras, aproveitando os conceitos de polimorfismo e de ligação dinâmica:

```
void rotacao_todas(Figura* v[], int numero, int angulo)
{
    for(int i = 0; i<numero; i++)
    v[i]->rotacao(angulo);
}
```

Para cada figura v[i], a função rotacao(), própria para o tipo actual do objecto, é chamada. O tipo actual do objecto v[i] não é conhecido na altura da compilação do programa.

Através do mecanismo de herança, as classes descendentes de Figura vão herdar os dados correspondentes ao centro e à cor da figura, não existindo assim a necessidade de definir dados para o centro e a cor sempre que se cria uma classe para representar uma nova figura. Também a

definição e o código correspondentes à função translacao() vão ser herdados pelas classes descendentes da classe figura. No que diz respeito aos métodos desenha() e rotacao(), o código terá que ser fornecido para cada classe criada para representar uma nova figura, pois estes métodos devem ser diferentes para cada figura devido à forma gráfica e ao algoritmo de rotação serem diferentes para objectos gráficos distintos.

Na definição da classe estão implícitas outras regras da linguagem C++, nomeadamente o conceito de construtor, o conceito de funções **inline**, o conceito de encapsulamento (características publicas (**public**) e características privadas (**private**)), e ainda a possibilidade de existirem funções com o mesmo nome, sendo estas diferenciadas através dos seus parâmetros, isto é, *overload* de funções (o caso dos dois construtores).

O **construtor** (neste caso, Figura) para uma classe é o método invocado automaticamente quando se quer crir um objecto (instância de uma classe). Enquanto que uma classe representa um tipo, um objecto representa uma entidade física cujo comportamento é definido pela sua classe. Além do construtor também poderá existir o **destrutor** (~Figura()), que é invocado pelos mecanismos internos da linguagem sempre que um objecto de uma determinada classe é destruído. O destrutor permite, por exemplo, libertar memória eventualmente alocada com o operador **new**.

As funções **inline** (Figura, translacao, centro_figura e cor_figura) obedecem às regras usuais dos tipos, ao contrário das *macros* comuns usados em C. Essas funções são expandidas na compilação em vez de serem invocadas quando em execução. A substituição **inline** de funções é particularmente importante no contexto da abstracção de dados e programação por objectos. Com estas técnicas de programação, as funções muito pequenas são muito comuns, e o aumento do número de chamadas a funções pode não implicar diminuição do desempenho.

Os dados escondidos (encapsulados) são os dados ou métodos que aparecem a seguir à palavra reservada **private**, sendo esses dados somente acedidos pelos métodos centro_figura e cor_figura. Isto evita o uso indevido desses dados por parte de programadores que usem esta classe para construir as suas hierarquias de classes.

As características públicas (definidas a seguir à palavra reservada **public**) de uma classe constituem a interface que objectos instanciados desta classe fornecem para o exterior.

Para definir uma figura em particular pode-se declarar que é uma figura e especificar as suas características específicas; por exemplo, a classe para representar um segmento de recta poderia ser definida da forma seguinte:

```
class Segmento_Recta :public Figura
{
private:
    //um segmento de recta é uma figura.
    ponto fim; // o ponto centro herdado de Figura representa o início do segmento de recta
public:
Segmento_Recta(ponto ini, ponto fi, int cor):Figura(inicio, cor), fim(fi){}
int operator == (Segmento_recta & oSeg) {
    return ( (inicio_seg()==oSeg.inicio_seg()) && (fim_seg() == oSeg.fim_seg()) );
}
ponto inicio_seg(){return centro_figura();}
```

```
ponto fim_seg(){return fim;}
void desenho();
void rotacao(int angulo);
}
```

A classe Segmento_Recta diz-se derivada da classe Figura, e a classe Figura é a classe base da classe Segmento_Recta. Uma classe derivada herda as características da sua classe base e também tem as suas características específicas. Por exemplo, a classe Segmento_Recta tem um membro "fim", em adição aos membros "cor" e "centro" que herdou da classe Figura. Além disso, tem duas novas funções para aceder aos pontos que definem o segmento de recta (inicio_seg() e fim_seg()). Outra função (int operator ==), exemplifica outra característica importante do C++ que é o *overloading* de operadores, neste caso trata-se da redefinição do operador relacional == de modo a que seja possível testar a igualdade de segmentos de recta, da mesma forma que se testa a igualdade de, por exemplo, variáveis inteiras.

Outra regra aqui detectada é a obrigatoriedade da invocação do construtor da classe base (caso esta o tenha) no construtor da classe derivada. Além disso, pode-se também invocar o contrutor de objectos membros da classe (neste caso o construtor de classe ponto).

Por fim, verifica-se que a nova figura "Segmento_Recta" foi adicionada sem haver a necessidade de modificar o código já escrito inclusivé o código para a função rotacao_todas(). A capacidade de estender um programa, adicionando novas variações de uma dada classe (isto é, adicionando novas classes derivadas de uma classe base) sem modificar o código já escrito, é uma grande vantagem face a outras técnicas de programação.

3.4. Windows e C++

O presente projecto envolve o desenvolvimento de aplicações em C++ para Windows; para tal recorre-se a mecanismos fornecidos pela linguagem C++ e interactuantes com o Windows, de forma a utilizar este ambiente numa filosofia de programação por objectos.

O problema é que a interface de programação de aplicações (API) do Windows inclui uma grande quantidade de funções e, além disso, define um elevado número de mensagens a que uma janela pode ter necessidade de responder [17].

Qualquer janela no Windows pertence a uma classe que comporta um conjunto de procedimentos associados a esta. Quando o Windows está a correr, são enviadas mensagens a uma janela sob a forma de chamada de um procedimento adequado da janela.

A tarefa básica de um procedimento da classe janela é identificar a mensagem que lhe foi enviada, processá-la e devolver o controlo (ao remetente da mensagem). Numa base ideal, todos os procedimentos da classe janela deviam estar habilitados a processar todas as mensagens possíveis do Windows; como isso é impossível na prática, a classe janela só fornece processamento dedicado para uma lista limitada de mensagens, re-enviando as restantes para uma função predefinida do Windows, *DetWndProc*, que contém o código, por defeito, para processar qualquer mensagem recebida.

A ideia natural conducente à criação de uma classe base em C++, para encapsular este comportamento da classe janela, é definir uma função membro correspondente a cada mensagem que a classe deve entender. Assim, uma classe derivada só necessitará de redefinir as funções correspondentes às mensagens que requerem um processamento especial. Contudo, esta ideia não é realizável, mais uma vez por factores práticos, pois o Windows define uma grande quantidade de mensagens possíveis, donde seria também necessário uma grande quantidade de funções distintas na classe base, que facilmente excederia a capacidade do compilador de C++.

Outra possibilidade consiste em reproduzir o comportamento do Windows, isto é, derivar classes em C++ que possam anular a função *DispatchMessage* do Windows. As mensagens seriam então recebidas inalteradas, da mesma forma que são recebidas numa aplicação em C, devendo o programador saber quais as mensagens a processar e quais as que não deveria processar, necessitando, portanto, de familiarizar-se com o significado dos parâmetros associados com as mensagens a serem processadas.

Uma ideia alternativa corresponde a uma solução de compromisso entre as duas hipóteses apresentadas acima. Esta ideia envolve a definição de funções virtuais, associadas com as mensagens mais comuns, e a permissão de aceder a mensagens subjacentes.

As implementações deste último tipo necessitam de algum processamento prévio à chamada da função associada com a mensagem, pois é necessário seleccionar a função associada à mensagem; por exemplo, à mensagem de escolha de um item de um menu está associada uma função que é invocada sempre que um menu é escolhido. No entanto, estas implementações escondem alguma complexidade do Windows tornando as mensagens mais abstractas (invocação de uma função específica do evento de escolha de um item de um menu) e, portanto, de mais fácil compreensão. Outra vantagem destas implementações é que escondem detalhes do tráfego das mensagens adjacentes (neste caso uma mensagem adjacente seria, por exemplo, o identificador do item escolhido no menu).

3.4.1. Win ++

A decisão de utilizar o Windows em conjunto com uma linguagem de programação orientada para objectos, levou à utilização de uma biblioteca de classes disponíveis comercialmente: o Win++ [18] fornece uma hierarquia de classes para o desenvolvimento de aplicações para o Windows, usando as facilidades de programação orientada para objectos, através do uso da linguagem C++.

A utilização do Win++ implicou um estudo da biblioteca de classes e do seu enquadramento no âmbito do desenvolvimento de aplicações para o Windows. De seguida descreve-se sucintamente esta biblioteca de classes, bem com a estrutura das aplicações que usam o Win++ como base; outros detalhes serão apresentados ao longo da descrição do *software* desenvolvido.

O objectivo do Win++ é fornecer uma classe para quase todas as entidades que o Windows trata como objectos, especialmente as entidades a que se tem acesso com um *handle*. Por exemplo, para o *display* de janelas existe uma classe base e uma série de classes derivadas para as formas de janelas mais comuns. Então, em vez de usar *handles* e invocar as funções do Windows, cria-se uma instância de uma classe, ou de uma classe derivada adaptada para o efeito, e chama-se a função membro fornecida pela classe do Win++. Para os restantes casos, e quando tal é necessário, o Win++ permite chamar directamente funções do Windows. Verifica-se, portanto, que o Win++ usa a terceira ideia expressa atrás.

3.4.2. Arquitectura do Win++

A figura 5 mostra uma parte da hierarquia de classes do Win++, excluindo-se as classes não utilizadas neste trabalho, de modo a simplificar a apresentação.

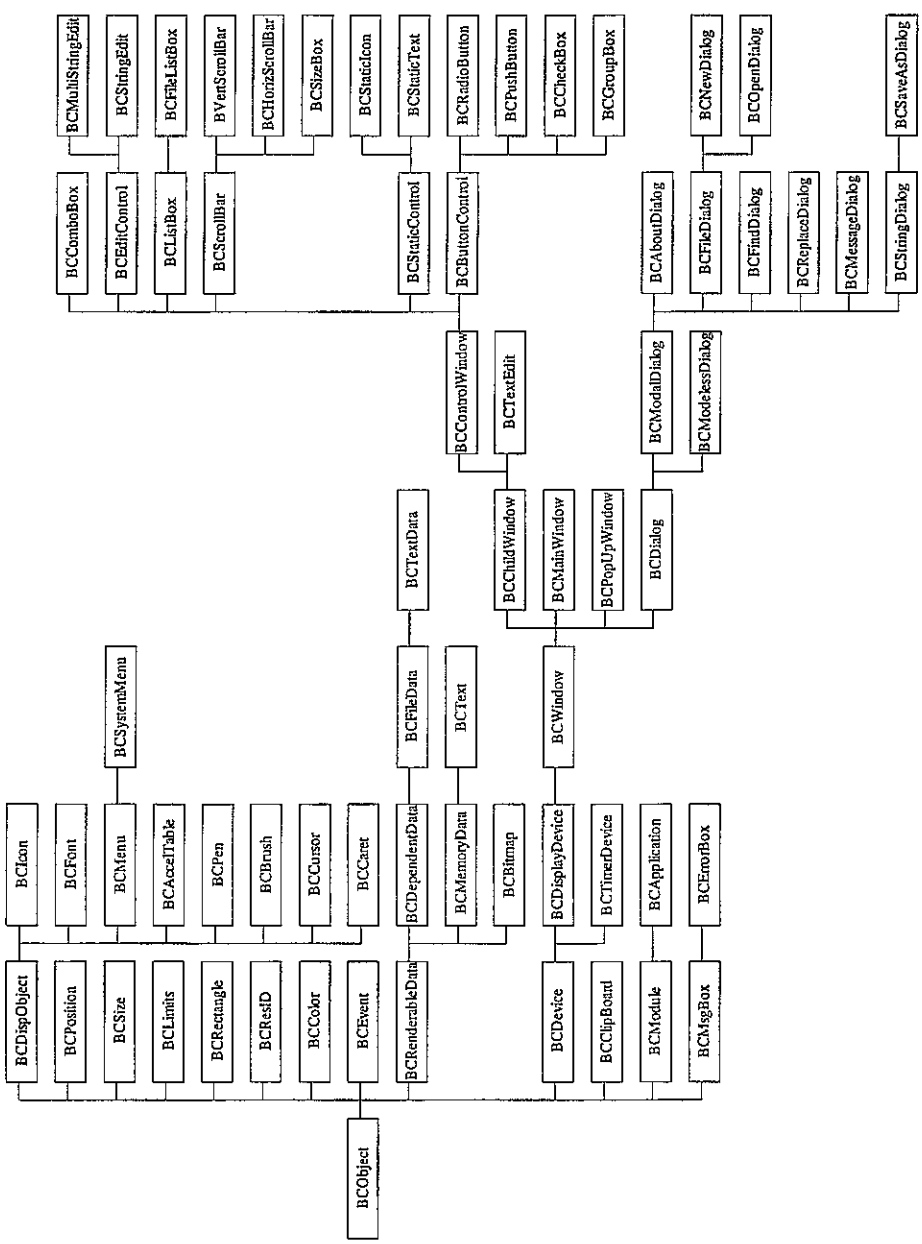


Fig. 5 - Hierarquia de classes utilizadas neste trabalho (classes do Win++)

O Windows é um sistema gerido por eventos, e passa os eventos para as aplicações sob a forma de mensagens notificadas. O Win++ define funções virtuais nas classes representativas dos objectos que recebem as mensagens, e chama-as internamente sempre que certos tipos de mensagens ocorrem. A estas funções de suporte de eventos, o Win++ passa um ponteiro para um objecto do tipo *BCEvent*, que contém mais informação acerca do evento. Este objecto, do tipo *BCEvent*, contém toda a informação passada pelo Windows para a mensagem e, em alguns casos, também inclui ponteiros para objectos relacionados com a mensagem (por exemplo botões ou menus).

O Win++ não tem funções individuais de suporte a todos os eventos possíveis; para obviar a esta limitação, fornece, na classe representativa da janela, duas funções (*UserEvt* e *OtherEvent*) para processar, entre outros, os eventos definidos pelo utilizador.

A apresentação da hierarquia (figura 5) tem por finalidade mostrar ao leitor as relações de herança entre as várias classes do sistema, facilitando assim a exposição do *software* desenvolvido neste trabalho. A importância de algumas destas classes na estrutura do *software*, principalmente nos módulos correspondentes à interface gráfica, é de tal forma acentuada que é necessário explicar com certo detalhe algumas das características dessas classes. Essa explicação será apresentada nos capítulos seguintes, quando tal for considerado necessário.

3.4.3. Estrutura das aplicações baseadas no Win++

A classe *BCApplication* é o ponto de partida para qualquer aplicação desenvolvida com base nas classes do Win++. Esta classe contém informação identificadora da aplicação (nomeadamente o seu *instance handle*, que identifica qualquer aplicação a correr no Windows), um ponteiro para os argumentos da linha de comando e outra informação de alto nível associada com a aplicação.

O código para a função *WinMain* está definido no Win++ de tal forma que, nessa função, é criada uma instância da classe *BCApplication* e é invocada internamente a função *StartUp* pertencente à classe *BCApplication*. Recordamos que a função *WinMain* é o ponto de entrada para todas as aplicações para ambiente Windows.

As funções *StartUp* e *ShutDown* são declaradas, mas não definidas, na classe *BCApplication*. O programador deve portanto codificar cada uma destas funções, as quais são invocadas internamente pelo Win++ quando a aplicação é iniciada (*StartUp*), e quando a aplicação é terminada (*ShutDown*). Em *StartUp* deve-se proceder à inicialização global da aplicação, nomeadamente deve criar-se uma instância da janela de aplicação *BCMainWindow* e, em seguida, invocar a função membro *Run*, que é responsável pelo processamento das mensagens.

Antes da aplicação terminar, a função membro *ShutDown* é invocada de dentro da função *Run*. Este procedimento dá a oportunidade ao programador de "apagar" informação, nomeadamente para libertar alguns recursos do Windows que tenham sido alocados pela aplicação.

Uma vez em execução, o programa constitui um suporte às mensagens originadas (a partir dos dispositivos de entrada, rato e teclado) por outras aplicações, pela própria aplicação ou pelo próprio Windows. O trabalho do programador é o de criar objectos que captem essas mensagens. Nesse processo há que criar, principalmente, funções membro para suporte de eventos, funções essas que substituam as funções fornecidas por defeito pelo Win++, para as mensagens que explicitamente se querem suportar.

Capítulo 4
A interface do PLC virtual

Depois desta fase introdutória passar-se-á à apresentação da interface gráfica bem como de alguns detalhes da sua implementação que são essenciais para a compreensão dos mecanismos que o Windows juntamente com o Win++ oferecem de modo a facilitar o desenvolvimento de interfaces gráficas.

O modelo do PLC virtual, que vai ser analisado posteriormente, pode esquematizar-se através da figura 6. Como se pode verificar, o PLC virtual comunica com o utilizador essencialmente à custa de editores, nomeadamente o editor de programação e o editor de estímulos. Em execução, o simulador lê as entradas do PLC virtual através do editor de estímulos, ou do editor interactivo, simula o programa presente no editor de programação, e permite visualisar os resultados da simulação através do editor de estímulos, ou do editor interactivo, ou ainda através duma janela do PLC real.

Devido à forte componente interactiva presente no PLC virtual, a construção desses editores que exteriorizam o PLC virtual deve torna-los amigáveis e de rápida aprendizagem, e fornecer uma interface *point-and-click* composta por menus e ícones de modo a facilitar a interacção com o PLC virtual.

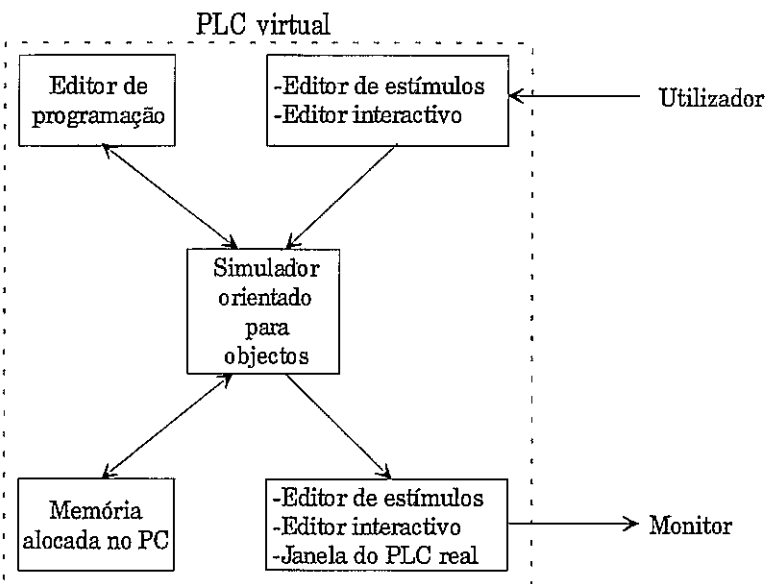


Fig. 6 - Modelo para a arquitectura do PLC virtual

Uma melhor concretização do que se exporá nos próximos capítulos pressupõe uma leitura prévia do Manual de Utilização da aplicação PLC virtual, que se encontra em apêndice.

4.1. Implementação de objectos gráficos de interface com o Win++

Como já foi referido, o Windows fornece um conjunto de objectos gráficos de interface. Alguns desses objectos, por exemplo, *bitmaps*, menus ou ícones, podem ser definidos como recursos dentro de uma aplicação. Um recurso representa então um objecto de interface do Windows, que pode ser incluído numa aplicação sem haver a necessidade de o gerar através de código na linguagem base. Os recursos são portanto desenhados e especificados fora do código da aplicação, e posteriormente são adicionados ao código compilado do programa de modo a criar um ficheiro executável para o ambiente Windows.

Existem ferramentas, nomeadamente o Borland Workshop [19], que permitem gerar todos os recursos para uma aplicação, possibilitando também a sua gravação num ficheiro de recursos (.rc). Cada recurso, presente no ficheiro de recursos, é identificado com um número único (ID).

Para construir a a interface entre os recursos da aplicação e o código fonte (.cpp), o Win++ fornece a classe *BCResID*. Um objecto, instanciado dessa classe, definido para um determinado recurso permite aceder a todas as suas características, sendo assim possível construir um do Win++ para representar esse recurso.

Depois de definidos os recursos, o ficheiro de recursos é compilado pelo compilador de recursos (Resourse Compiler [19]) dando origem a um ficheiro de recursos compilado (.res), sendo posteriormente ligado ao código objecto (.obj) da aplicação pelo mesmo compilador, de modo a criar um ficheiro executável (.exe) para o Windows. Na figura 7 é apresentada a sequência de acções necessária para o desenvolvimento de uma aplicação para ambiente Windows. No topo dos vários rectângulos aparece a extensão dos ficheiros definidos no interior do rectângulo

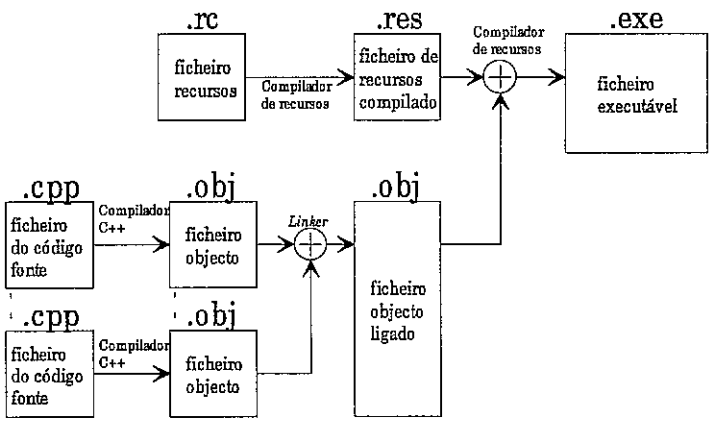


Fig. 7 - Construção de uma aplicação para ambiente Windows

Como já foi referido o Windows define vários objectos gráficos de interface. Seguidamente apresenta-se a forma como alguns desses objectos são construídos usando as classes fornecidas pelo Win++, nomeadamente *bitmaps*, cursores, menus e caixas de diálogo.

Bitmaps

A classe de suporte a recursos do tipo *bitmap* é a classe *BCBitmap*. Esta permite criar os objectos que suportam os *bitmaps* definidos no ficheiro de recursos, como se pode ver no código para a criação do *bitmap* usado para representar a opção Contacto Aberto, e cujo o ID é IDR_BMP_OPEN_CONTACT (Apêndice A, ponto A.3.3).

Cursores

Para a implementação de cursores o Win++ fornece a classe *BCCursor*, que suporta os cursores definidos no ficheiro de recursos. Como já referimos é possível definir a aparência dos cursores, sob a forma de *bitmaps*, utilizando o Workshop.

Para a criar e inserir um objecto do tipo Cursor numa janela é apresentado o exemplo seguinte, que cria o cursor identificado por IDR_EDITOR_PROG_INACTIVE_CURSOR (que representa o cursor para sinalizar a opção Inactiva), e instala-o na janela através da função membro da janela, *SetCursor*.

```
inactiveCur = new BCCursor(BCResID(IDR_EDITOR_PROG_INACTIVE_CURSOR));
if (inactiveCur)
    SetCursor(inactiveCur);
```

Menus

No ficheiro de recursos são também definidos os menus para as várias janelas do PLC virtual. A classe tipo para a criação dos menus é a classe *BCMenu* que, além de permitir a criação de menus, também permite inserir ou modificar items ou sub-menus, incluindo os menus baseados em *bitmaps*. A listagem seguinte exemplifica a criação, a inserção e a instalação de um item (tipo *bitmap*) no menu da janela de aplicação, definido no ficheiro de recursos sob o ID de IDR_MAIN_MENU.

```
// Cria o objecto menu, com os items definidos no ficheiro de recurso sob o identificador IDR_MAIN_MENU
opMenu = new BCMenu(BCResID(IDR_MAIN_MENU));
// Cria um objecto do tipo BCBitmap, com o bitmap cujo ID é IDR_BMP_OPEN_CONTACT
inputOpenBitmap = new BCBitmap(BCResID(IDR_BMP_OPEN_CONTACT));
if(inputOpenBitmap)// insere o item IDM_INPUT_OPEN na posição POS_INI_MENU+1
    opMenu->Insert(IDM_INPUT_OPEN, inputOpenBitmap, POS_INI_MENU+1);
...
SetMenu(opMenu);        // instala o menu na janela
```

Caixas de diálogo

As classes base para a criação de caixas de diálogo são *BCModalDialog* e *BCModelessDialog*. O *template* para a caixa de diálogo é definido no ficheiro de recursos, sendo a caixa de diálogo criada com a chamada ao seu construtor, o qual deve invocar a função para

criação da janela da caixa de diálogo (*Create*). No caso de classes derivadas de *BCModalDialog*, essa função não regressa enquanto não for invocada a função para *terminus* da caixa de diálogo, *EndDlg*; para as classes derivadas de *BCModelessDialog* a função *Create* regressa após a criação.

As selecções são executadas por manipulação de objectos gráficos denominados por controlos. O editor de caixas de diálogo do Workshop permite desenhar a caixa de diálogo incluindo o posicionamento dos controlos. Por exemplo, a criação de uma caixa de diálogo para a edição dos parâmetros dos objectos na janela da simulação interactiva (Apêndice A, ponto A.5.4.3.3), baseou-se na classe seguinte:

```
class IntSimulEditButtonDlg : public BCModalDialog
{
    InputOutputRadioButton *opInOutButton;      // ponteiro para o objecto a editar os parâmetros
    C20RealSimulInteractive *opIntSimulDlg;     // ponteiro para a janela de simulação interactiva
    BCRadioButton *opRadioLabel;                // ponteiro para o botão Etiqueta
    BCRadioButton *opRadioAddress;              // ponteiro para o botão Endereço
    BCPushButton *opOKButton;                   // botão de controlo OK
    BCStringEdit *opLabelEdit;                  // para edição da string da etiqueta
    BCStringEdit *opAddressEdit;                // para edição da string do Endereço
    BCStaticText *infoText;                     // para escrita do texto de informação
protected:
    WORD Initialize(VOID);
    VOID ButtonSnglClk(BCEvent *opEvt);
    ...
public:
    IntSimulEditButtonDlg (InputOutputRadioButton *opInOutButton, C20RealSimulInteractive *opIntSimulDlg,
                           BCResID &oResID);
    ...
};
```

O processo de criação de uma nova caixa de diálogo comporta várias fases.

Em primeiro lugar deve-se definir o *template* da caixa de diálogo, com todos os controlos necessários, utilizando por exemplo o utilitário Workshop. O Win++ fornece uma classe tipo para cada um dos controlos definidos pelo Windows [10]. Cada classe baseia-se na classe base *BCControlWindow*, a qual define as características dos objectos do tipo controlo.

Depois, é necessário derivar uma classe de uma das duas classes base (*BCModalDialog* ou *BCModelessDialog*) do Win++.

Finalmente, devem criar-se ponteiros para os objectos do Win++ que representam os tipos de controlos inseridos no *template* da caixa de diálogo, e redefinir algumas funções virtuais para atendimento a eventos. No exemplo anterior encontram-se objectos do tipo *BCRadioButtom* (classe para representar controlos do tipo *radio buttons*), *BCPushButton* (classe para representar *push buttons*), *BCStringEdit* (classe-tipo dos controlos para editar *strings*), *BCStaticText* (classe-tipo dos controlos para escrita de texto). As funções virtuais para atendimento a eventos, neste caso, são só duas: *Initialize* (para criar os objectos do tipo controlo) e *ButtonSnglClk* (que é invocada quando um botão é seleccionado). Para melhor compreender estas duas funções, apresenta-se parte do código com alguns comentários:

```
WORD IntSimulEditButtonDlg:: Initialize(VOID)
{
    // criação do objecto do tipo BCRadioButton para representar o botão Etiqueta
    // IDC_LABEL_INT_SIMUL é o identificador do botão Etiqueta no template da caixa de diálogo
    opRadioLabel = new BCRadioButton(BCResID(IDC_LABEL_INT_SIMUL), this);
    // criação do objecto do tipo BCRadioButton para representar o botão Endereço
    opRadioAddress = new BCRadioButton(BCResID(IDC_ADDRESS_INT_SIMUL),this);
    if ( opInOutButton->GetShowLabel() )
        opRadioLabel->SetState(TRUE);       // a etiqueta é a seleccionada
    else
        opRadioAddress->SetState(TRUE);     // o endereço é o seleccionado
    // criação do controlo para edição da etiqueta
    opLabelEdit = new BCStringEdit(BCResID(IDC_LABEL_INT_SIMUL_UPDATE),this);
    opLabelEdit->SetText(opInOutButton->GetLabel());// inserção da etiqueta do objecto no controlo
    opLabelEdit->SetFocus();                        // focus para este controlo
    opLabelEdit->SetCharLimit(MAX_LABEL_SIZE);      // máximo de caracteres da etiqueta
    ...
    // cria o controlo para escrita de informação
    infoText = new BCStaticText(BCResID(IDC_INFO_STATIC_TEXT),this);
    if(opInOutButton->IsInput())
        infoText->SetText("Entrada");               // o objecto editado é do tipo Entrada
    else
        infoText->SetText("Saída / Ponto interno"); // o objecto editado é do tipo Saída / Ponto interno
    return(0);
}
```

```
VOID IntSimulEditButtonDlg:: ButtonSnglClk(BCEvent *opEvt)
{
CHAR caBuf[120];
    switch (opEvt->GetControlID()) {            // detecção do botão seleccionado
        case IDOK:                              // o botão OK foi seleccionado
            opLabelEdit->GetText(caBuf, MAX_LABEL_SIZE);
            if (IsValidLabel(caBuf))            // verifica a validade da etiqueta introduzida
                opInOutButton->SetLabel(caBuf); // actualiza a etiqueta do objecto editado
            ...// o mesmo para o endereço
            EndDlg(TRUE);                       // termina a caixa de diálogo
        break;
        case IDCANCEL:                          // o botão CANCEL foi seleccionado
            EndDlg(FALSE);
        break;
        case IDC_LABEL_INT_SIMUL:               // o botão Etiqueta foi seleccionado
            opRadioLabel->SetState(TRUE);       // actualiza o botão Etiqueta
            opRadioAddress->SetState(FALSE);    // actualiza o botão Endereço
            opInOutButton->ShowLabel(TRUE);     // actualiza o objecto editado
        break;
        ...                                     // o contrário para o botão endereço
    }
}
```

O método a seguir para qualquer outra caixa de diálogo é o mesmo, podendo variar no tipo e quantidade de controlos inseridos na caixa de diálogo, e podendo ainda haver a necessidade de redefinição de outras funções virtuais a partir das classes antecessoras de *BCModalDialog* (*BCDialog* ou *BCWindow*), por forma a atender outro tipo de eventos.

4.2. Interface gráfica

Além dos objectivos específicos da interface do PLC virtual com o utilizador, foi também definido como objectivo que ela fosse gráfica e que permitisse uma aprendizagem simples e rápida, tal como acontece em interfaces gráficas de aplicações comerciais.

Para atingir esses objectivos especificaram-se à partida algumas características da interface, bem como os métodos a utilizar para atingir esses fins:

- a utilização da interface do Microsoft Windows, com a consequente utilização das entidades gráficas fornecidas por este, nomeadamente janelas, menus, ícones e caixas de diálogo;
- desenvolvimento de um editor de programação assente num editor essencialmente gráfico, em que os símbolos a inserir no editor são, na realidade, objectos de classes pré-definidas;
- desenvolvimento de um ambiente simulação adequado para cada modo de simulação implementado (simulação rápida, simulação interactiva e simulação em tempo real);
- contruir esses ambientes de modo a que selecções, instruções e operações sejam executadas com (e a partir de) menus, ícones e utilização do rato e/ou teclado.

A interface do PLC virtual com o utilizador pode resumir-se a quatro janelas principais (figura 8) que representam a base para a interface gráfica.

Como já foi referido a interface gráfica é constituída, essencialmente, pelos vários editores pertencentes ao modelo do PLC virtual. As classes que suportam as janelas dos vários são derivadas de classes do Win++ e, como tal, a comunicação entre o Windows e os editores (que são objectos instanciados dessas classes) é feita à custa das classes do Win++. Além dos vários editores, existe também a janela de aplicação que fornece alguns mecanismos de comunicação entre os editores.

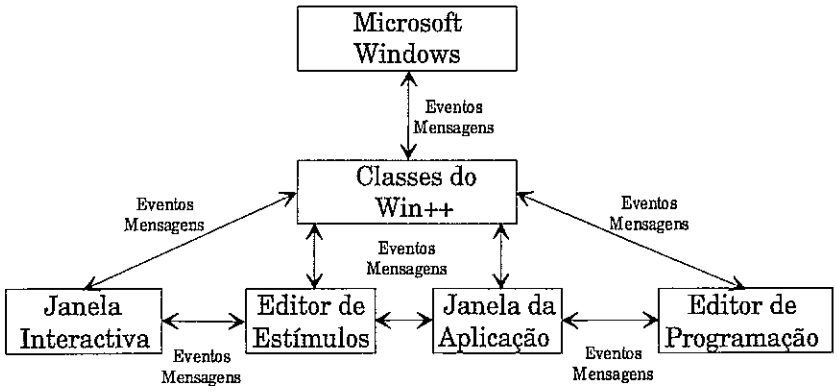


Fig. 8 - Comunicação entre as várias janelas do PLC virtual

A figura 8 mostra a interacção, quando em execução, das várias janelas do sistema PLC virtual com o Windows. Como se pode verificar, o Win++ serve de ligação entre o Windows e o código em C++ da aplicação.

Como já referimos atrás as mensagens enviadas pelo Windows são processadas pela biblioteca de classes do Win++, a qual internamente invoca funções virtuais de classes que pertencem à hierarquia de classes do próprio Win++. Deste processo são exemplos, entre outras, a classe base representativa da janela (*BCWindow*) e a classe base representativa de temporizadores (*BCTimerDevice*). O programador deriva, então, as suas classes das do Win++, por forma a representar os seus objectos, e redefine as funções virtuais para as mensagens que deseja processar. Desta forma é assegurado o fluxo de mensagens do Windows para a aplicação. Para a comunicação entre os diversos componentes do sistema são fornecidas funções membro às classes representativas dos vários objectos (neste caso, janelas), de modo a permitir a comunicação entre as várias janelas do PLC virtual.

Para facilitar a exposição sobre a implementação genérica da interface gráfica do PLC virtual, apresenta-se, no ponto seguinte, o exemplo da implementação da janela de aplicação.

4.3. A janela da aplicação

A janela de aplicação é o substrato em que as outras janelas (editores) assentam, pois é através de mecanismos de comunicação, implementados ao nível desta janela, que as outras comunicam entre si. Passaremos então a descrever o processo usado para implementar a janela de aplicação (ver também o Apêndice A, ponto A.3.1).

Por imposição do Win++, foi construída, por derivação da classe *BCMainWindow*, a classe *SimulC20Window* para representar a janela de aplicação do PLC virtual.

O ponto de entrada da aplicação é a função *StartUp*, na qual é criado um objecto (do tipo *SimulC20Window*). Em seguida, é invocado o método *Run* que vai gerir o ciclo de mensagens.

Para terminar a aplicação é necessário desenvolver o código para o função *ShutDown,* de modo a destruir o objecto que representa a janela de aplicação. Este método é invocado internamente pelo Win++ na sequência de comandos que permitem fechar a aplicação.

A classe *BCWindow,* que representa o objecto gráfico janela na interface Windows, define as características de uma janela, nomeadamente aquelas que permitem gerir a janela e os seus componentes. De entre essas características destacam-se as funções de gestão da janela e as funções de suporte a eventos. Todas as funções que providenciam a saída gráfica para a janela são fornecidas pela classe *BCDisplayDevice* e são herdados por classes descendentes de *BCWindow*. Todas as funções para suporte de eventos são declaradas virtuais na classe *BCWindow*, podendo ser redefinidas na classe janela do utilizador de modo a esta responder aos eventos de forma adequada. De entre as funções para gestão da janela, citamos funções para posicionar, esconder e minimizar a janela, para mudar/instalar menus, e para instalar cursores. Além disso, objectos instanciados de classes descendentes de *BCWindow* podem manter objectos para representar ferramentas de *display* como, por exemplo, pincéis (*brushes*), lápis (*pens*), *bitmaps* e tipos de caracteres (*fonts*).

A classe *SimulC20Window* contém a informação necessária para gerir todo o sistema, nomeadamente ponteiros para as diversas janelas do PLC virtual e outros para controlo da interface (e. g. , menus, *bitmaps* e aceleradores).

As funções de atendimento a eventos são declaradas na zona protegida da classe, e representam as redefinições das funções virtuais da classe *BCWindow*. A maneira de descrever, de uma forma mais exacta, as características fundamentais da classe representativa da janela de aplicação é a definição da própria classe. Convirá referir que, quer as definições das classes, quer o código das funções aqui apresentadas, não será completo, pois uma grande parte das funções e/ou dados não são relevantes para a compreensão do *software* desenvolvido.

```
class SimulC20Window : public BCMainWindow
{
    ...
    C20Omron *opC20Omron;              // ponteiro para o objecto que representa o PLC virtual
    ProgramEditor *opProgramEditor;    // janela de suporte ao editor de programação
    SimulationWindow *opSimulWindow;// janela de suporte à simulação,
    StatusWindow *opStatusWin;         // janela de informações
    EditWindow *opTextEditor;          // janela de um editor de texto
    BCAccelTable *opAccel;             // aceleradores para os menus da janela de aplicações
    BCMenu *opMenu;                    // menu da janela de aplicação
    MatrixCircuit *Data;               // objecto que suporta a grelha do programa
    WORD OptionSelection;              // identificador da opção seleccionada
    BCBitmap *OptionBitmap;            // bitmap representativo da opção seleccionada
    CHAR *OptionText;                  // nome da opção seleccionada
    BCBitmap *inputOpenBitmap, ..., *inputCloseBitmap;  // bitmaps de todas as opções
    BOOL LockOption;                   // sinaliza bloqueamento da opção seleccionada
protected:
    // redefinição das funções pré-definidas na classe BCWindow, pertencentes a classes acima na hierarquia
    // todas estas funções são invocadas internamente pelo Win++ quando ocorrem eventos específicos
    WORD Initialize(VOID);             // invocada para permitir a inicialização de dados, ou ferramentas
    VOID Paint(BCEvent *);             // invocada quando é necessário desenhar a janela
    VOID MenuSelect(BCEvent *);        // invocada quando um menu é seleccionado
    VOID MenuCommand(BCEvent *);       // invocada quando um menu é escolhido
    VOID ReSize(BCEvent *);            // invocada quando uma janela é redimensionada
    VOID Activation(BCEvent *);        // invocada quando uma janela é activada
    BOOL QueryEnd(BCEvent *);          // invocada antes de terminar uma aplicação
    VOID  GetMinMaxSizes(BCSize &,BCSize &);  // lê as dimensões mínimas e máximas da janela
public:
    SimulC20Window(VOID);                  // construtor da classe
    ~SimulC20Window(VOID);                 // destrutor da classe
    ...
};
```

No construtor da classe é invocada a função *Create* (definida em *BCWindow*), que cria a janela. Esta função invoca internamente a função virtual *Initialize* (que está redefinida na classe *SimulC20Window*) antes de ser feito o *display* da janela no monitor. A função *Initialize* é geralmente usada para criar ou inicializar objectos de controlo e também certos atributos que necessitem de um *handle* janela como, por exemplo, menus ou botões. Neste caso, é também usada para criar as outras janelas, pois estas, sendo janelas filhas da janela de aplicação, necessitam obviamente de uma janela mãe para serem criadas.

A seguir apresenta-se uma parte do código da função *Initialize*:

```
WORD SimulC20Window :: Initialize(VOID)
{
    opMenu = new BCMenu(BCResID(IDR_MAIN_MENU));                    // criação do menu da janela
    // criação da janela de suporte ao editor de estímulos
    opSimulWindow =new SimulationWindow(this,BCPosition(SIMUL_WINDOW_X,SIMUL_WINDOW_Y));
    // criação da janela de suporte ao editor de programação
    opProgramEditor = new ProgramEditor(this,BCPosition(EDIT_WINDOW_X,EDIT_WINDOW_Y));
    // criação da janela de informações
    opStatusWin = new StatusWindow(this,BCPosition(STATUS_WINDOW_X,STATUS_WINDOW_DY));
    // criação do editor de texto
    opTextEditor = new EditWindow(this);
    opC20Omron = new C20Omron(this, opSimulaChild);                 // cria o plc virtual
    // criação e inserção no menu dos items tipo bitmap para representação das opções
    inactiveBitmap = new BCBitmap(BCResID( IDR_BITMAP_INACTIVE ));  // representa a opção Inactiva
    if(inactiveBitmap)
    opMenu->Insert(IDM_INACTIVE,inactiveBitmap,POS_INI_MENU);       // insere o bitmap no menu
    ...
    opAccel = new BCAccelTable(BCResID(IDR_ACCEL_TABLE));           // criação dos aceleradores
    GetApp()->SetAccel(this,opAccel);                               // insere os aceleradores na aplicação
    Data = opChild->GetCircuit();          // ponteiro para a matriz de suporte do programa ladder
    return(0);
}
```

Como se pode verificar, são criadas instâncias das classes representativas dos vários objectos usados na aplicação, nomeadamente a janela de suporte ao editor de programação (*opProgramEditor*), a janela de suporte à simulação (*opSimulWindow*) e o objecto do PLC virtual (*opC20Omron*).

A comunicação entre a janela de aplicação e o utilizador é garantida à custa de menus (as opções são menus em forma de *bitmaps*). O Win++ define duas funções de atendimento a eventos gerados pela escolha ou selecção de menus, e que são *MenuCommand e MenuSelect,* respectivamente. Este último pode ser usado para enviar mensagens para o utilizador sobre o item do menu seleccionado, como se pode ver no código seguinte, o qual envia para a janela de informações a mensagem "Permite inserir um símbolo do tipo Saída Normalmente Aberta" (Apêndice A, ponto A.3.4).

```
VOID SimulatorC20Window::MenuSelect(BCEvent *opEvt)
{
    switch (opEvt->GetMenuID()) { // descobre o item do menu seleccionado
        ...
        case IDM_INPUT_OPEN: // informa na janela de informações sobre o item do menu
            opStatusWin->Commentary("Permite inserir um símbolo do tipo Saída Normalmente Aberta");
        break;
    }
}
```

Através do objecto do tipo *BCEvent*, passado internamente pelo Win++ para a função virtual *MenuSelect,* é possível obter informação da mensagem enviada pelo Windows,

nomeadamente o identificador do menu seleccionado. Como se pode ver, quando o menu da opção Saída Normalmente Aberta é seleccionado, é enviado para a janela de informação uma mensagem.

Na função *MenuCommand* são tomadas as acções de resposta à escolha de um menu. Neste caso, é o menu da janela de aplicação. O código seguinte ilustra o processo de resposta à escolha de alguns menus.

```
VOID SimulC20Window :: MenuCommand(BCEvent *opEvt)
{
    CHAR caBuf[81];
    switch (opEvt->GetMenuID()){ // descobre o item do menu seleccionado
        case IDM_OPEN:    // menu para carregar um ficheiro do disco
            if (Data->OpenFile()) {  // invoca a função OpenFile, de modo a carregar o arquivo para o editor
                sprintf(caBuf,"Simulc20H - %s",Data->GetFileName() );    // lê o nome do ficheiro
                SetTitle(caBuf);          // modifica o título da janela de acordo com o nome do ficheiro aberto
            }
        break;
        case IDM_COPY:    // menu para copiar parte do programa para o clipboard
            if(opProgramEditor->GetIsSelect())    // verifica se existem células seleccionadas
                Data->Copy(opProgramEditor->GetInitSelect(), opProgramEditor->GetEndSelect());    // copia
        break;
        case IDM_INPUT_OPEN:  // menu para seleccionar a opção Contacto Aberto
            OptionSelection = IDC_OPEN_CONTACT;  // identifica a opção Contacto Aberto
            OptionBitmap = inputOpenBitmap;          // bitmap correspondente à opção
        break;
        ...
    }
}
```

Verifica-se que, para cada item do menu, existe uma ou mais acções tendo em vista fornecer a funcionalidade desejada através dos menus fornecidos. Existem acções que actuam sobre a própria janela de aplicação (IDM_INPUT_OPEN - escolha da opção Contacto Normalmente Aberto) e acções que invocam funções de outros objectos, nomeadamente IDM_COPY e IDM_OPEN. A escolha do menu IDM_COPY permite copiar para o *clipboard* as linhas de programa seleccionadas na janela do editor de programação.

Em seguida apresentam-se outros métodos de resposta a eventos para a janela de aplicação, nomeadamente: eventos de *Paint*, que ocorrem quando o Windows necessita de fazer o *upgrade* gráfico da janela (por exemplo, em consequência de sobreposição de janelas); eventos de *ReSize* quando é feito o redimensionamento da janela; evento de *Activation* da janela, que ocorre quando a janela fica activa ou inactiva.

```
VOID SimulC20Window :: Paint(BCEvent *opEvt)
{
    BCBrush oBrush(BC_LIGHT_BRUSH);          // cria um objecto do tipo pincel
    BCBrush *opOldBrush = SetBrush(&oBrush);  // instala o novo pincel na janela
    PutRect(GetUserRect(),BC_FILL_IT);        // desenha um rectângulo na janela
    SetBrush(opOldBrush);                     // instala o pincel anterior
}
```

Todo o código (objectos e funções membro) utilizado na função *Paint* é fornecido pelo Win++. Nesta função é criado um objecto do tipo *BCBrush* (para representar um pincel), que é instalado na janela de aplicação. Em seguida, é desenhado um rectângulo a cheio, cujas dimensões correspondem às dimensões da área de utilizador da janela de aplicação.

A função *Activation* é invocada internamente quando a janela é activada (ou desactivada), por exemplo através do *click* do rato sobre a janela. Como a janela do editor de programação se encontra sobre a janela de aplicação deve-se, então, detectar uma activação da janela de aplicação de modo a dar o *focus* de entrada à janela do editor de programação.

```
VOID SimulC20Window :: Activation(BCEvent *opEvt)
{
    if (opEvt->IsActivated())          // verifica se a janela foi activada e não desactivada
        opProgramEditor->SetFocus();   // dá o focus à janela do editor de programação
}
```

A função *ReSize* é invocada sempre que há variação no tamanho da janela (de aplicação). As janelas do editor de programação e de informações dependem das dimensões da janela de aplicação e, por este motivo, devem ser redimensionadas sempre que há variações na janela de aplicação.

```
VOID SimulC20Window::ReSize(BCEvent *opEvt)
{
        // redimensiona a janela do editor de programação
        opProgramEditor->SetWindowRect(BCRectangle(BCPosition(EDIT_WINDOW_X,EDIT_WINDOW_Y),
                                GetUserRect().LowerRight()-BCPosition(0,STATUS_WINDOW_DY+2)));
        // redimensiona a janela de informações
        opStatusWin->SetWindowRect(BCRectangle(GetUserRect().LowerLeft(),
            BCPosition(STATUS_WINDOW_X,STATUS_WINDOW_DY), GetUserRect().LowerRight()));
}
```

Da descrição anterior, sobre a janela de aplicação, constata-se a simplicidade com que se constroem os mecanismos de interacção com o utilizador usando os conceitos de herança e de funções virtuais do C++. De facto, bastou derivar uma classe da classe base representativa da janela de aplicação do Win++ para se obterem, por herança, as funções de gestão da janela (incluindo as funções para escrita e desenho na janela), e ter acesso a todas as funções virtuais de suporte a eventos. Algumas dessas funções foram redefinidas de modo a responder aos respectivos eventos de uma forma apropriada à interface a implementar.

Como é conhecido, toda a gestão da janela, e dos dispositivos de entrada, é feita internamente pelo Windows e, neste caso, é comunicada à aplicação através da invocação, pelo Win++, de funções virtuais. De notar que praticamente todo o código apresentado depende do código do Win++, código esse, reutilizado sem haver necessidade de qualquer modificação.

Capítulo 5
Editores gráficos do PLC virtual

5.1. Editor de programação

Como já foi referido, o editor de programação deve permitir de uma forma simples e intuitiva a escrita de programas em linguagem *ladder*.

A janela do editor de programação (Apêndice A, ponto A.3.1) está contida na janela de aplicação, sendo a sua classe representativa derivada da classe *BCChildWindow* (classe base para todas as janelas filhas, incluindo as janelas para representar controlos). A janela do editor contém a grelha de suporte ao programa e, como tal, tem a responsabilidade de gerir todas as acções que o utilizador efectuar sobre a grelha.

5.1.1. Símbolos (objectos) *ladder*

A linguagem *ladder* é composta por um conjunto de instruções identificadas por símbolos gráficos. Algumas das relações entre esses símbolos gráficos podem ser expressas através de hierarquias. Por isso foram criadas classes tipo, ligadas hierarquicamente por mecanismos de herança, para representar cada um dos símbolos *ladder* (instruções) implementados neste trabalho e que se encontram descritos no Apêndice A, ponto A.3.2.

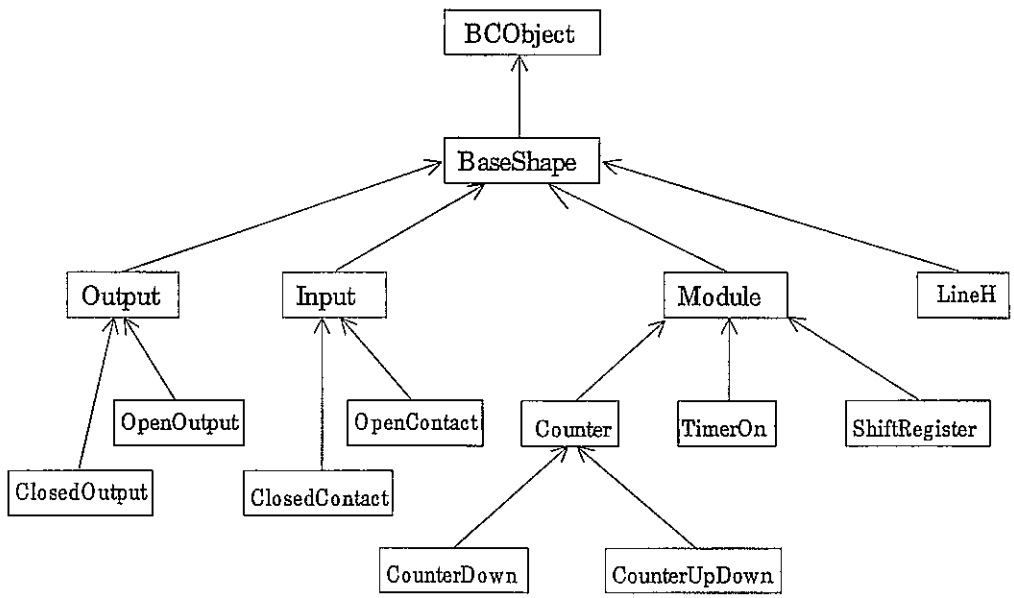


Fig. 9 - Hierarquia das classes representativas dos símbolos *ladder* implementados

Todos os símbolos *ladder* são objectos (instâncias) de classes terminais pré-definidas (ClosedOutput, TimerOn, ...), que pertencem à hierarquia (figura 9) cuja classe base é a classe *BaseShape*. Esta é uma classe abstracta que descreve as características que todas as classes descendentes devem possuir. Por exemplo, cada objecto (representando um símbolo) tem um nome, conhece a sua localização numa grelha bidimensional e, além disso, deve conhecer os objectos que lhe estão ligados para poder realizar as tarefas de simulação que lhe estão destinadas (capítulo 6). Cada objecto deve possuir funções membro que serão invocadas para realizar as operações de desenho (para se desenhar) e de simulação (quando o comportamento do símbolo está a ser simulado).

A definição de uma classe abstracta permite, de uma forma consistente, o envio de mensagens para objectos instanciados de classes pertencentes à hierarquia. De facto, uma das utilidades das classes abstractas reside no facto de objectos com um antepassado comum poderem ser encarados como objectos do mesmo tipo.

A seguir apresentam-se os dados e as funções membro mais importantes da classe *BaseShape*:

```
class BaseShape : public BCObject
{
protected:
    static ProgramEditor *opProgramEditor;  // ponteiro para a janela de desenho do símbolo
    CHAR *objectName;           // nome do objecto (símbolo)
    CHAR **objectLabel;         // etiquetas que indicam posições na memória do PLC virtual, necessárias
                                // para a execução da instrução subjacente ao objecto
    WORD opValue;               // para armazenar um valor numérico associado com a instrução
    BCPosition XYLocation;      // coordenadas da localização na matriz
    BOOL lineUp;                // sinalização da existência de uma ligação vertical
    BaseShape *leftShape,*upShape,*downShape;   // ponteiros para objectos ligados à esquerda, acima e abaixo
    int **memoryLocation;       // ponteiros para a memória (palavras) referenciadas pelas etiquetas
    int *maskBit;               // localizações dos bits nas respectivas palavras
    BOOL flagSimulation;        // flag usada durante a simulação, para sinalizar que o objecto já foi simulado
    virtual VOID OwnerSimulation(BOOL inSimul)=0;  // função que processa a sua entrada e actua de acordo com
                                                   // a sua própria função
    ...
public:
    virtual BOOL Simul(VOID);                          // função pública para gestão da simulação
    virtual VOID ResetValuesToFirstSimulation(VOID);   // inicializa os dados do objecto usados na simulação
    virtual VOID ResetToOtherSimulation(VOID);         // inicia os dados de controle da simulação
    virtual VOID Draw(VOID);                          // função pública para desenhar o objecto dentro da grelha
    virtual VOID UnDraw(VOID);                         // função pública para apagar o objecto dentro da grelha
    virtual BCBitmap * GetShapeBitmap(void) {return NULL;}   // devolve um ponteiro para o bitmap do símbolo
    ...
};
```

Posta esta definição da classe *BaseShape*, por exemplo para definir uma classe para representar símbolos do tipo Saída Normalmente Aberta basta criar um novo *bitmap* para a representação gráfica, e escrever o código correspondente a duas funções virtuais, ao construtor e destrutor da classe; todas as outras características são herdadas da classe *BaseShape*.

```
class  OpenOutput : public Output
{
protected:
    static BCBitmap *opShapeBitmap;     // representa o bitmap para representação gráfica
    virtual BOOL OwnerSimulation(BOOL valIn);
public:
    OpenOutput(BCPosition & oPos, CHAR *opLab = "" );
    ~OpenOutput();
    virtual BCBitmap * GetShapeBitmap(VOID) {return opShapeBitmap;}    // devolve o bitmap
    ...
};
```

Para representar um temporizador, basta derivar uma classe de *Module* e acrescentar as particularidades que distinguem objectos deste tipo de outros objectos.

```
class TimerOn : public Module
{
protected:
    static BCBitmap *opShapeBitmap;     // representa o bitmap para representação gráfica
    WORD timerNumber;   // número do temporizador
    WORD oTime;         // tempo de temporização
    WORD oTimeInSteps;  // número de varrimentos de simulação necessários para realizar o tempo de temporização
                        // (oTimeInSteps  = oTime / stepPeriod)
    WORD numberStep;    // varrimento actual (contagem a partir de zero)
    WORD stepPeriod;    // periodo de varrimento
    virtual BOOL OwnerSimulation(BOOL valIn);   // realiza a simulação do temporizador
    ...
public:
    virtual VOID Draw(VOID);                                // desenha o temporizador
    virtual VOID ResetTimer(VOID);                          // inicializa o temporizador
    virtual VOID ResetValuesToFirstSimulation(VOID);        // inicializa os dados para a simulação
    virtual BCBitmap * GetShapeBitmap(VOID) {return opShapeBitmap;}
    ...
};
```

Como se depreende da definição das classes, as características dos símbolos podem ser divididas em dois grupos: a parte gráfica (que o utilizador manipula) e a parte funcional (que gere a funcionalidade do símbolo no diagrama *ladder*).

Neste capítulo, que versa os Editores Gráficos, não serão analisadas as características funcionais, mas somente as gráficas, pois são estas as manipuladas directamente pelo editor de programação.

Desenho dos símbolos

A aparência gráfica de cada símbolo é forçosamente diferente da de outros, sendo o aspecto gráfico de um símbolo garantido por um *bitmap* definido para o efeito. Isto permite que o aspecto de um símbolo seja facilmente alterado sem haver a necessidade de modificar o código. Como objectos do mesmo tipo têm a mesma aparência, não faz sentido haver um *bitmap* para cada

objecto, pelo que se define como **static** o objecto do tipo *BCBitmap* em classes terminais (classes que se podem instanciar dando origem a objectos *ladder*). Tal procedimento possibilita que o mesmo *bitmap* seja visto por todas as instâncias dessa classe.

Posto isto, a função *Draw* limita-se a colocar o *bitmap* na posição respectiva da grelha e, em seguida, escrever os parâmetros visíveis do objecto sobre o *bitmap*. Para isso, usam-se as funções membro de *ProgramEditor*, que foram herdadas de *BCDisplayDevice*, para a colocação de *bitmaps* e escrita de *strings* na janela.

A seguir mostra-se o código da função *Draw* para a classe *BaseShape* que, por mecanismos de herança, serve os objectos do tipo entradas e saídas. De assinalar que o *bitmap* do símbolo é devolvido pela função virtual *GetShapeBitmap*, pelo que a função *BaseShape::Draw* pode servir vários objectos gráficos, já que *GetShapeBitmap* vai devolver o *bitmap* correspondente ao símbolo que está a ser desenhado.

```
VOID BaseShape :: Draw(void)
{
    if ( CanDraw() ){              // verifica se o objecto está na parte visível da janela
        // cálculo da posição inicial em pixeis
        BCPosition oPos( XYLocation.X() * CELL_WIDTH, XYLocation.Y() * CELL_HEIGHT);
        // colocação do bitmap na janela
        opProgramEditor->PutBitmap( oPos, BCSize(CELL_WIDTH, CELL_HEIGHT), GetShapeBitmap());
        // escrita da etiqueta sobre o objecto
        opProgramEditor->PosStr(oPos+BCPosition(CELL_WIDTH/2, 2), ShapeLabel[0]);
    }
}
```

5.1.2. Suporte do editor

O suporte para o editor de programação é uma matriz bidimensional de ponteiros para objectos do tipo *PointOfGrid* (figura 10). Cada objecto deste tipo representa um nodo (ponto de ligação), nodo este que identifica o conteúdo da célula à sua esquerda. A razão de ser da célula à esquerda, e não à direita, tem a ver com o processo de simulação, que é feito a partir de objectos terminais e direccionado para o início de continuidade do diagrama *ladder* (capítulo 6).

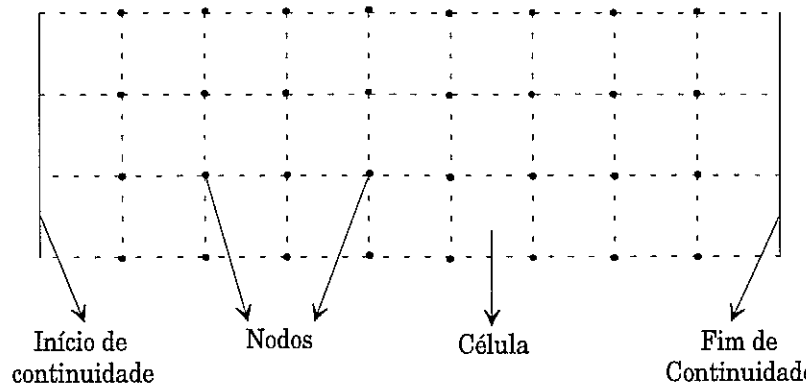


Fig. 10 - Grelha de suporte ao programa *ladder*

A forma mais expedita de construir o suporte para o editor é criar uma classe que caracterize por completo a grelha. Essa classe é a classe *MatrixCircuit* que é derivada da classe *BCFileData* do Win++. Esta classe (*BCFileData*) associa um objecto que contém dados com um ficheiro, fornecendo funções membro para as operações usuais com ficheiros, nomeadamente operações de *New*, *Open*, *Save*, *SaveAs* e *Close*. A classe *MatrixCircuit* define os dados e as funções membro para todo o tipo de operações necessárias para suporte, em memória, de programas em linguagem *ladder*.

```
class MatrixCircuit:public BCFileData
{
    PointOfGrid ***m;                   // ponteiro para a matriz bidimensional de ponteiros
    WORD MatrixDx , MatrixDy;           // dimensões da matriz
    ProgramEditor *opProgramEditor;     // ponteiro para a janela do editor de programação
    ...
public:
    MatrixCircuit( ProgramEditor *Win, WORD Dx, WORD Dy, MatrixCircuit *Circ = NULL);
    // apaga todos os objectos no rectângulo iniciado na célula initPos e terminado na célula endPos
    BOOL ClearCircuit(BCPosition initPos, BCPosition endPos);
    BOOL InsertLadderObject(BCPosition oPos, BaseShape *Shape); // insere um objecto na posição oPos da matriz
    BOOL InsertLineV(BCPosition Pos);                   // insere uma linha vertical na posição oPos
    BOOL Delete(BCPosition oPos);                       // apaga o elemento da posição oPos
    BYTE CanInsert(BCPosition oPos, BaseShape *Base);   // verifica se pode inserir um dado elemento
    BYTE CanDelete(BCPosition oPos);                    // verifica se pode apagar um dado elemento
    BOOL Replace(VOID);                                 // substitui um objecto já existente numa dada posição
    // funções virtuais redefinidas
    // invocada internamente quando é necessário apagar todo o programa, por exemplo numa operação de Load
    BOOL Clear(BOOL blDelete = TRUE);
    BOOL Load(BCFile *opFile);          // redefinida para carregar um programa para a memória
    BOOL Save(BCFile *opFile);          // redefinida para gravar um programa para disco
    // para criar objectos a partir do ficheiro do arquivo
    BOOL CreateElement(BCPosition oPos, symbol obj, char *lab[], WORD val=0);
    VOID Draw(VOID);                    // desenha todos os objectos da grelha
    // copia os objectos entre as posições pIni a pEnd para o clipboard
    BOOL Copy(BCPosition pIni, BCPosition pEnd);
    BOOL Paste(BCPosition oPaste);      // coloca os objectos que estão no clipboard na posição oPaste
    BOOL InsertNewLine (WORD oLine);    // insere uma linha a partir da linha oLine
    // métodos usados para a simulação
    BOOL MakeConections(VOID);          // inicializa as ligações entre os objectos na grelha
    VOID ClearConections(VOID);         // destrói todas as ligações entre objectos
    VOID ResetAllTimers(VOID);          // inicializa todos os objectos do tipo temporizadores
    // inicializa todos os dados de controle, de modo a simular outro varrimento do programa
    VOID ResetForOtherSimulation(VOID);
    ...
};
```

Esta classe mantém uma matriz bidimensional de ponteiros para objectos do tipo *PointOfGrid*. Cada objecto deste tipo representa um nodo da grelha, nodo este que contém dados sobre o objecto *ladder* presente na célula correspondente. Esses dados incluem um ponteiro para o próprio objecto, e informação sobre o tipo de objecto e sobre a ligação vertical do próprio nodo. As células, por definição, representam os locais possíveis para a inserção de símbolos,

funcionando cada nodo como um meio de acesso ao objecto representativo do símbolo, em memória. Assim, definiu-se a classe representativa de cada nodo:

```
class PointOfGrid
{
    BOOL Down;              // ligação vertical ao nodo da linha seguinte
    BaseShape *LeftShape;   // objecto ladder ligado à esquerda (na célula da esquerda)
    BYTE CellState;         // identificador do tipo do objecto ladder presente na célula
public:
    ...
    VOID SetLeftShape(BaseShape *Left){ LeftShape = Left; }
    BaseShape * GetLeftShape(){return LeftShape;}
};
```

A geração da matriz é dinâmica, o que permite criar programas maiores sem haver constrangimentos de memória. O construtor da classe cria a matriz, inicializa os outros dados da classe e invoca o construtor de *BCFileData*, o qual inicializa, por exemplo, a extensão dos ficheiros (extensão por defeito).

```
MatrixCircuit::MatrixCircuit(ProgramEditor *Win, WORD Dx, WORD Dy, MatrixCircuit *Circ):
                BCFileData(Circ, (BCWindow *)Win, ".c20","",BC_KEEP_DATA | BC_DONT_COPY_DATA)
{
    ...
    // alocação de memória para a matriz
    m = new PointOfGrid **[Dx];
    for(int i = 0; i < Dx; i++)
        m[i] = new PointOfGrid *[Dy];
    for(i = 0; i < Dx; i++)
        for(int j = 0; j< Dy; j++)
            m[i][j] = new PointOfGrid();
}
```

Depois de criado um objecto do tipo *MatrixCircuit,* este fica com a capacidade de executar as operações necessárias para suportar em memória toda a estrutura associada à grelha.

Para a inserção na grelha fornecem-se duas funções, sendo uma para inserir uma ligação vertical (*InsertLineV*), e a outra para inserir um objecto *ladder* (*InsertLadderObject*). Por exemplo, para a função *InsertObjectLadder* o código é apresentado a seguir. A função *CanInsert* foi desenvolvida para "obrigar" o utilizador a obedecer a algumas regras, por exemplo, a inserção de saídas somente na última coluna e a não possível inserção de símbolos tipo módulo na primeira ou última coluna.

```
BOOL MatrixCircuit::InsertLadderObject(BCPosition oPos, BaseShape *Shape)
{
BYTE CanIns = CanInsert(oPos,Shape);  // verifica se pode inserir o objecto na posição oPos - testa algumas regras
    if( CanIns == CAN_INSERT_ELEMENT ){  // pode inserir o objecto na grelha
        GetXY(oPos)->SetLeftShape(Shape); // GetXY(oPos) - devolve um ponteiro para o nodo correspondente
        SetModified(TRUE);                // sinaliza mudança no programa para efeitos de gravação em ficheiro
        return TRUE;
    }
    if( CanIns == CELL_IS_NOT_EMPTY )     // célula não está vazia
        if( Replace() ) {  // pode-se fazer a substituição da célula
            BaseShape *oShape = GetPosXY(oPos);  // objecto presente na posição oPos
```

```
        delete oShape;                          // apaga o objecto existente na posição oPos
        GetXY(oPos)->SetLeftShape(Shape);       // insere o novo objecto no nodo correspondente
        SetModified(TRUE);
        return TRUE;
    }
    else {
        delete Shape;           // apaga o objecto Shape, pois não foi inserido na grelha
        return FALSE;
    }
}
```

5.1.3. Janela e ferramentas do editor de programação

A janela do editor de programação constitui a interface do utilizador para um objecto do tipo *MatrixCircuit*, e fornece as ferramentas para o "desenho" do programa.

A janela do editor é filha da janela de aplicação, encontrando-se graficamente sobre essa mesma janela. O editor possui uma grelha constituída por células, na qual vão ser colocados símbolos gráficos. A interligação adequada desses símbolos (na horizontal e na vertical) permite representar um programa em linguagem *ladder*.

O editor fornece ferramentas para inserir e apagar símbolos na grelha, e permite utilizar os comandos tradicionais de Copiar, Cortar, Apagar e Colocar. Além disso, permite também a edição dos símbolos, de modo a modificar os parâmetros que os definem ou caracterizam.

A classe que define todas as características do editor é a seguinte:

```
class ProgramEditor : public BCChildWindow
{
    SimulC20Window *opPrt;          // ponteiro para a janela de aplicação
    BOOL IsSelect;                  // sinaliza a selecção para copy, clear ou cut
    BOOL GridOn;                    // sinaliza a visibilidade ou invisibilidade da grelha
    BCPosition SelectedCell;        // identifica a célula seleccionada em cada instante
    BCPosition initSelect, endSelect;   // célula inicial e final que indicam o rectângulo de células seleccionado
    BCSize SizeCell;                // tamanho em pixeis de cada célula
    MatrixCircuit *Cir;             // objecto do tipo MatrixCircuit que suporta o programa em memória
    BCCursor *deleteCur;            // cursor tipo borracha, para sinalizar a opção Borracha
    BCCursor *inactiveCur;          // cursor para sinalizar a acção de editar parâmetros do objecto
    BCCursor *insertCur;            // cursor para sinalizar a acção de inserção de objectos na grelha
    BCCursor *vertLineCur;          // cursor para sinalizar a acção de inserção de linhas verticais
    BCPen *editClearPen;            // pincel para apagar símbolos
    ...
protected:
    // redefinição de funções de atendimento a eventos, pertencentes a classes acima na hierarquia
    WORD Initialize(VOID);          // atende o evento de inicialização
    VOID Paint(BCEvent *);          // desenha completamente a janela, invocada num evento de desenho
    VOID Activation(BCEvent *opEvt);    // invocada quando a janela é activada ou desactivada
    VOID VertScroll(BCEvent *opEvt);    // invocada quando é actuado o scroll bar vertical
    VOID HorizScroll(BCEvent *opEvt);   // invocada quando é actuado o scroll bar horizontal
    VOID KeyDown(BCEvent *opEvt);   // invocada quando uma tecla é premida
    VOID MouseDblClk(BCEvent *);    // atende eventos de duplo click do rato
    VOID MouseMove(BCEvent *);      // atende o evento de movimento do rato
    VOID MouseDrag(BCEvent *);      // atende o evento de drag do rato
    VOID MouseDown(BCEvent *);      // invocada quando o botão do rato é premido
    VOID MouseUp(BCEvent *);        // invocada quando o botão do rato é levantado
```

```
public:
    ...
    BOOL WorkElement(BCPosition oPos, WORD Element);  // insere um símbolo na posição oPos da grelha
    BCPosition FindCell(BCPosition oPos);        // devolve a célula da grelha que corresponde à posição do rato oPos
};
```

A descrição dos dados considerados mais relevantes pode ser vista na parte privada da classe. De entre os dados destacam-se os vários tipos de cursores para sinalização do utilizador sobre o estado do editor de programação, e o objecto (*Cir*) de suporte à grelha do programa.

No método *Initialize,* de que é apresentada uma pequena parte, é inicializada a maioria dos objectos que a janela mantém (cursores, pincéis e o objecto de suporte à grelha).

```
WORD ProgramEditor :: Initialize(VOID)
{
    ...
// cria o cursor cujo template está definido no ficheiro de recursos - ID = IDR_EDITOR_PROG_DELETE_CURSOR
    deleteCur = new BCCursor(BCResID(IDR_EDITOR_PROG_DELETE_CURSOR ));   // cria o cursor deleteCur
    // cria o objecto Cir com NUMBER_HORIZONTAL_CELL x NUMBER_VERTICAL_CELL nodos
    Cir= new MatrixCircuit(this, NUMBER_HORIZONTAL_CELL, NUMBER_VERTICAL_CELL, NULL);
    editClearPen = new  BCPen(BC_SOLID_LINE, 2, BCColor(BC_WHITE));      // cria o pincel editClearPen
    return(0);
}
```

Desenho da janela de programação

A janela de programação usa um objecto (*Cir*) do tipo *MatrixCircuit* para suportar em memória o programa escrito em linguagem *ladder*, o qual consiste de um conjunto de símbolos gráficos dispostos de forma adequada sobre uma matriz bidimensional de células. O desenho da grelha bidimensional é feita no método *Paint*, no qual também é enviada uma mensagem de *Draw* para o objecto grelha (*Cir*) de modo a desenhar todo o programa na janela do editor de programação. A sequência da passagem de eventos, por forma a redesenhar o programa no editor, é apresentada a seguir: o processo é desencadeado pelo Windows, que sinaliza a necessidade de redesenho da janela através de um evento de redesenho; em seguida, na função *Paint* é enviado um evento de *Draw* ao objecto *Cir* que, por sua vez, o enviará a cada objecto representativo de cada símbolo na grelha.

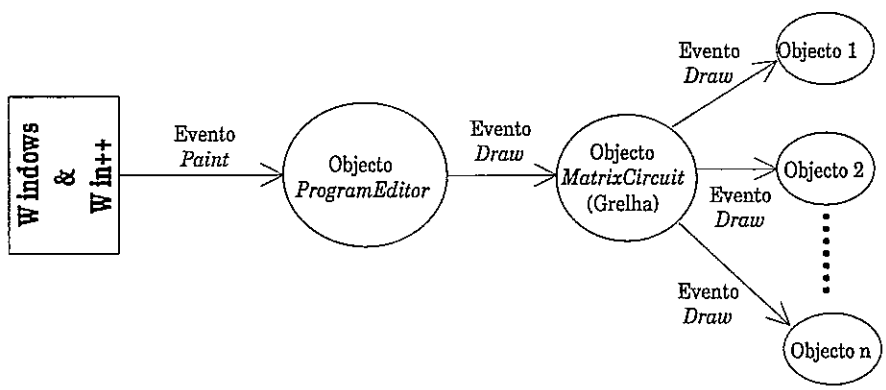


Fig. 11 - Sequência de eventos para o desenho da janela do editor de programação

```
VOID ProgramEditor :: Paint(BCEvent *opEvt)
{
    Clear();          // limpa a janela  - Clear() é herdado da classe BCDisplayDevice
    if(GridOn){       // desenho da grelha
        ...
    }
    Cir->Draw();      // desenha o programa na grelha
}
```

Inserção e apagamento de símbolos

A inserção de um símbolo *ladder* pelo utilizador é executada em duas fases: a primeira corresponde à escolha do símbolo a inserir (Apêndice A, ponto A.3.6.1) por selecção do símbolo no menu da janela de aplicação; a segunda parte corresponde à inserção do objecto na grelha, e é executada premindo o rato sobre a célula onde se deseja inserir o símbolo. Do ponto de vista de programação, basta responder ao evento de rato premido, de forma adequada, isto é, basta redefinir a função virtual *MouseDown*:

```
VOID ProgramEditor:: MouseDown(BCEvent *opEvt)
{
 BCPosition oPos = opEvt->GetPos();// posição do rato enviada no objecto opEvt
 SelectedCell = FindCell(oPos);          // célula onde se quer inserir
 UpdateCellInfor();                      // actualiza a célula na janela de informações
 // cria e insere na grelha o objecto ladder correspondente à opção seleccionada
 WorkElement(SelectedCell, opPrt->GetOptionSelection()); // GetOptionSelection() - devolve a opção seleccionada
 ...
}
```

O processo de criação e inserção de um novo símbolo *ladder* na grelha de suporte do programa é o ilustrado na figura 12.

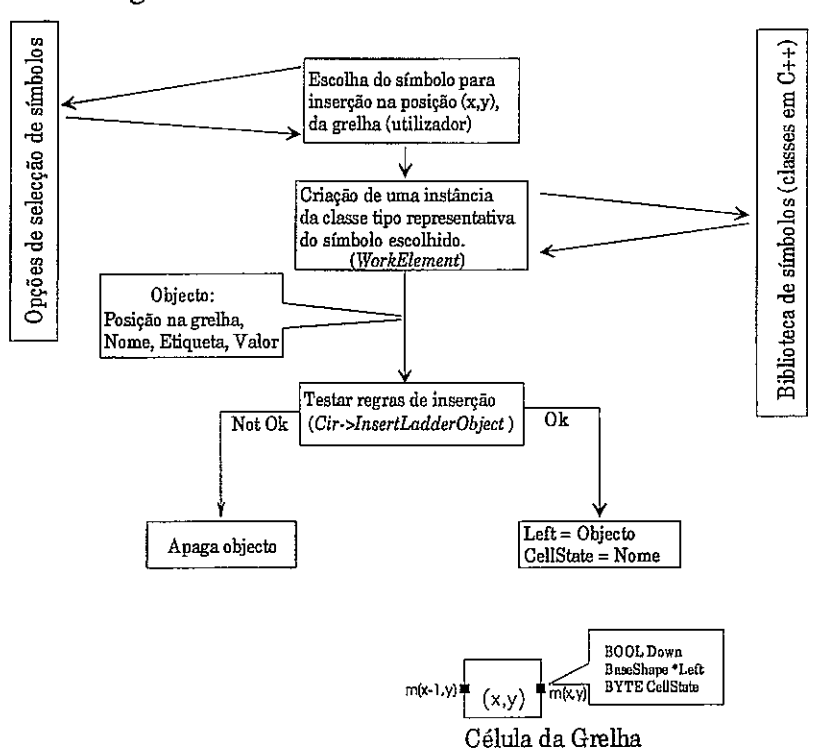


Fig. 12 - Processo de inserção de um objecto na grelha

De notar que m(x,y) corresponde ao nodo que identifica completamente a célula (x,y).

Na função *WorkElement* processam-se as acções de inserção ou apagamento de objectos da grelha, na posição seleccionada. O parâmetro *Element* permite descodificar a acção pretendida, isto é, verifica qual o tipo de símbolo que o utilizador quer inserir, ou então se o utilizador quer apagar algum símbolo.

```
BOOL ProgramEditor :: WorkElement(BCPosition oPos, WORD Element)
{
    BaseShape *Shape = NULL;
    switch (Element) {                          // identifica a opção seleccionada
        case IDC_OPEN_CONTACT:                  // para inserção de um Contacto Normalmente Aberto
            Shape = new OpenContact(oPos);      // cria uma instância da classe tipo de Contacto Normalmente Aberto
                if( Cir->InsertLadderObject(SelectedCell, Shape) )  // o objecto foi correctamente inserido na grelha
                    Shape->Draw();                                  // desenho do objecto
            break;
        ...
        case IDC_DELETE:                    // a opção seleccionada é a Borracha
            Cir->Delete(SelectedCell);      // apaga o objecto presente sobre a célula seleccionada
        break;
    }
}
```

Para a inserção de um símbolo, é criado um objecto para representar esse símbolo e, em seguida, é invocada a função (*InsertLadderObject*), pertencente ao objecto *Cir*, para inserir o objecto na grelha. No caso da opção ser de apagamento, apenas se invoca o método *Delete* da grelha.

Edição de parâmetros dos símbolos

Como já foi referido, um símbolo *ladder*, além do seu aspecto gráfico, suporta um conjunto de parâmetros, que necessitam obviamente de ser associados ao objecto correspondente. Para a edição de um símbolo o utilizador deve premir duas vezes o rato sobre o símbolo, sendo então criada uma de dois tipos de caixas de diálogo. Estas caixas de diálogo são representadas pelas classes *ElementEdit* ou *LabelEdit*, consoante se trate de um símbolo tipo módulo ou de símbolos tipo entrada e/ou saída (Apêndice A, ponto A.3.6.3). Do ponto de vista de programação, basta responder ao evento de duplo *click* do rato da forma conveniente, o que se consegue através da redefinição da função *MouseDblClk*, que é invocada internamente pelo Win++ quando o rato é premido duas vezes seguidas sobre a janela do editor de programação.

```
VOID ProgramEditor ::MouseDblClk(BCEvent *opEvt)
{
    BCPosition oPos = opEvt->GetPos();      // retira, do evento, a posição do rato
    SelectedCell = FindCell(oPos);          // descobre a célula correspondente à posição do rato
    BaseShape *shapeToEdit = Cir->GetPosXY(SelectedCell); // objecto ladder a editar
    if(shapeToEdit->IsModule() ){                         // o símbolo a editar é do tipo módulo
        ElementEdit oDlg( BCResID(IDD_ELEMENT_EDIT), this, shapeToEdit);   // cria a caixa de diálogo
        if (oDlg.GetResult())               // se foi premido o botão de OK na caixa de diálogo
            shapeToEdit->Draw();        // redesenha o símbolo já com os novos parâmetros
        return;
```

```
    }
    if( shapeToEdit->IsSingleCell() ){      // o símbolo é do tipo entrada ou saída
        // cria uma caixa de diálogo cujo template se encontra no ficheiro de recursos sobre o identificador
        // IDD_EDIT_SHAPE, e cujo tipo é dado por uma classe LabelEdit definida para suportar este template
        LabelEdit oDlg( BCResID(IDD_EDIT_SHAPE), this, shapeToEdit);
        if (oDlg.GetResult())
            shapeToEdit->Draw();
    }
    ...
}
```

Cursores

Existem quatro tipos de cursores para melhor informar o utilizador sobre o tipo de acção esperada quando o botão do rato é premido. Os cursores fornecidos são: cursor para sinalizar a opção Inactiva (que permite editar um símbolo); cursor para apagamento de símbolos; cursor para sinalização de ligação vertical; e um cursor para sinalização de inserção de um símbolo na grelha. A mudança dos cursores é feita na função redefinida *MouseMove*, que é invocada sempre que há movimento do rato dentro da janela do editor de programação.

```
VOID ProgramEditor:: MouseMove(BCEvent *opEvt)
{
WORD Selection =opPrt->GetOptionSelection();
    ...
    if ( Selection == IDC_INACTIVE ) {      // se a opção seleccionada é a opção Inactiva
        SetCursor (inactiveCur);            // instala o cursor que sinaliza a opção Inactiva
        return;
    }
    if ( Selection == IDC_DELETE ){         // se a opção seleccionada é a Borracha
        SetCursor (deleteCur);
        return;
    }
    if ( Selection == IDC_LINEV ){          // se a opção seleccionada é a ligação vertical
        SetCursor (vertLineCur);
        return;
    }
    // instala o cursor, para sinalizar que se pode inserir o símbolo correspondente à selecção efectuada
    SetCursor (insertCur);
}
```

5.1.4. Armazenamento em disco

Para a gravação em disco e posterior carregamento de programas *ladder* foram utilizadas os tradicionais menus de "Abrir", "Gravar" e "Gravar Como".

A decisão sobre a estrutura dos ficheiros a usar para armazenamento dos programas *ladder* recaiu sobre um formato do tipo texto. Sendo, então, o arquivo uma tradução fiel da grelha do editor, esta forma de arquivo dos programas em modo texto permite a escrita dos programas com um simples editor de texto, possibilitando ao utilizador a escrita de programas noutro computador mesmo que não possua o Windows.

5.1.4.1. Formato do arquivo

De seguida apresenta-se uma breve descrição do formato do ficheiro para armazenamento.

Cada símbolo tem um conjunto de parâmetros que o identificam e que permitem a sua reconstrução posterior. Esses parâmetros, na sequência em que aparecem no ficheiro, são:

- Um nome (obrigatório)
- Três etiquetas, no máximo (poderá não ter nenhuma)
- Um valor (optativo)
- Um sinalizador da existência de uma ligação vertical (optativo)

A posição do símbolo na grelha é guardada através da sua posição no ficheiro.

As regras que definem o formato de programa *ladder*, quando armazenado no disco, são:

- Cada linha é inicializada com o símbolo "{" e é terminada com o símbolo "}"
- A separação entre células é feita através do símbolo ","
- Linhas vazias são representadas pela sequência "{/" nº de linhas vazias "}"
- O nome do símbolo é obrigatório, sendo os outros parâmetros opcionais
- Cada parâmetro é separado por um ou mais espaços e/ou *tabs*
- O valor é representado pela sequência "(#" valor ")"
- A ligação vertical, para o nodo imediatamente abaixo, é sinalizada pelo símbolo "!", no fim de todos os outros parâmetros

O número de etiquetas depende do tipo de símbolo representado; por exemplo, um Registo de Deslocamento necessita de duas etiquetas que identificam o endereço das palavras inicial e final.

Exemplos de nomes para identificação de alguns símbolos:

LN	representa uma ligação horizontal
IO	representa um Contacto Normalmente Aberto
IC	representa um Contacto Normalmente Fechado
OO	representa uma Saída Normalmente Aberta
UDC	representa um Contador Ascendente/Descendente
...	
SFT	representa um Registo de Deslocamento

5.1.4.2. Gravação dos arquivos

O fluxograma das acções envolvidas na gravação apresenta-se na figura 13, podendo constatar-se que a grelha é percorrida desde a posição (0,0) até ao fim, registando-se em disco os parâmetros que identificam cada objecto. A não existência de um símbolo implica somente a

escrita do separador ",". A estrutura dos ficheiros de arquivo, corresponde a uma estrutura equivalente (com pequenas diferenças) daquela utilizada em [20] para o armazenamento de ficheiros de programas.

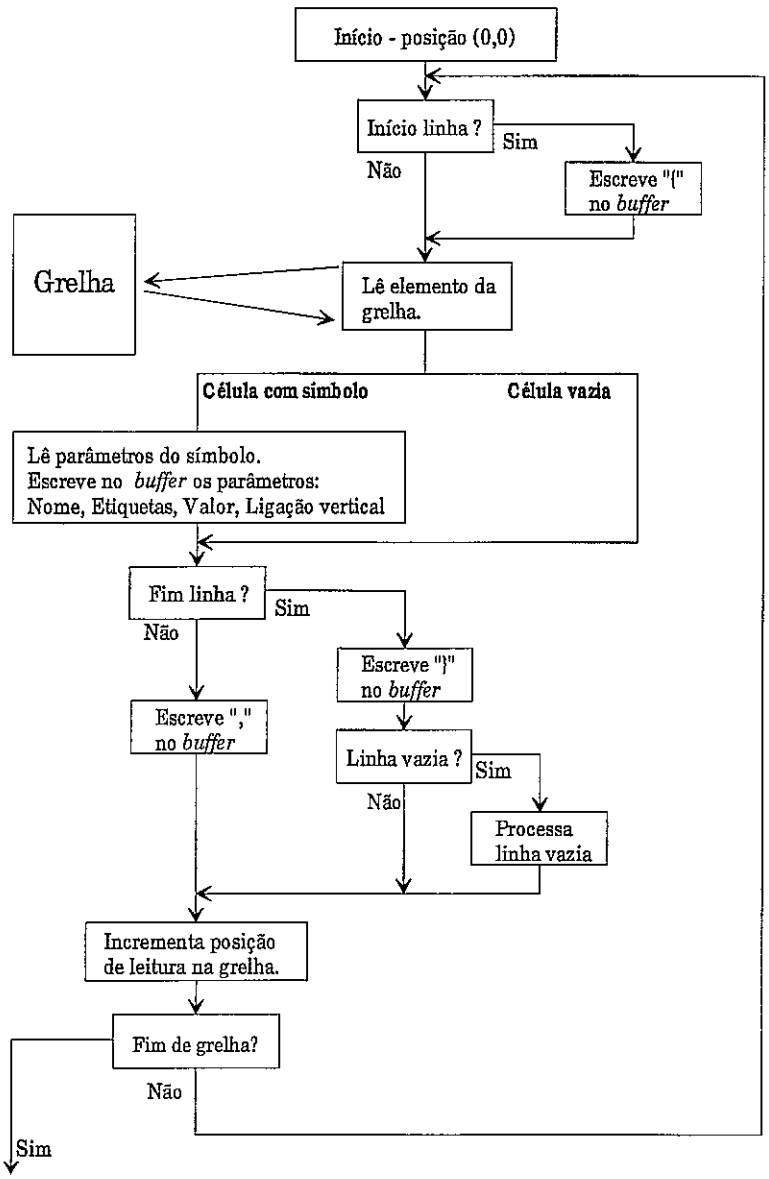


Fig. 13 - Processo de gravação em disco de um programa

A seguir, apresenta-se um exemplo das duas possíveis descrições para uma parte de um programa, ou seja, a forma gráfica e a forma textual (figura 14). De recordar que uma linha em branco é representada pela sequência {/1}.

```
{IC IR_10001 , LN , TON TC_000 (#250), LN , LN , LN , LN , OO IR_10000}
{IC IR_10001 , IO IR_10000 , LN , LN , TON TC_002 (#250), LN , LN , LN , OO IR_10001}
{IC IR_10002 , IO IR_10003 , TON TC_003 (#15000), LN , LN , LN , LN , OO IR_10002}
{/1}
```

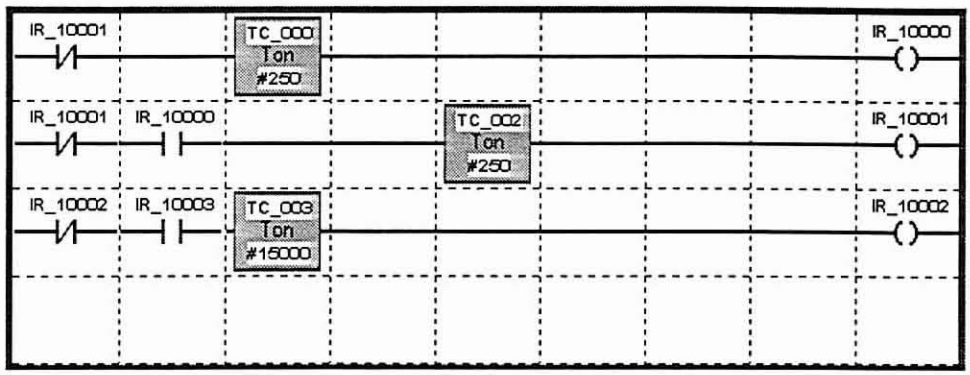


Fig. 14 - Diagrama *Ladder* correspondente à descrição textual do arquivo anterior

5.1.4.3. Leitura dos arquivos

A sequência de acções, isto é, a sequência de invocação de funções para carregar um programa armazenado em disco para o formato da linguagem *ladder* do editor de programação esquematiza-se na figura 15.

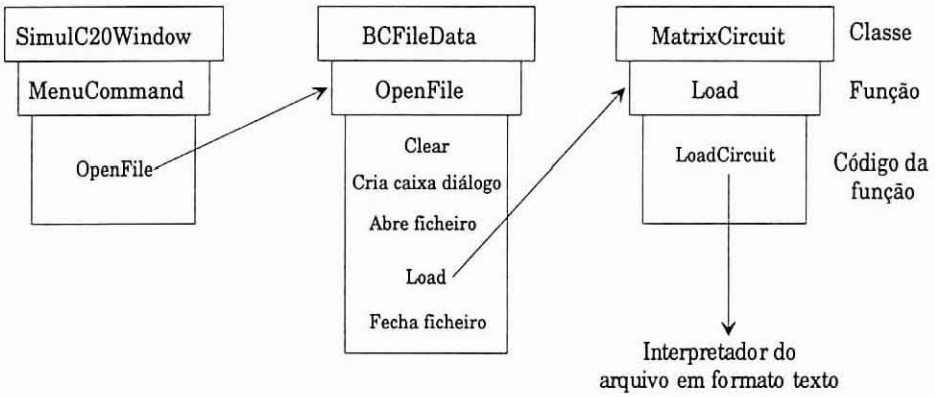


Fig. 15 - Sequência de invocação de funções para carregar um programa

Na sequência da escolha do menu "Abrir" da aplicação é invocada (internamente pelo Win++) a função de suporte a eventos de menus, *MenuCommand*, herdada de *BCWindow* e redefinida em *SimulC20Window*. Em *MenuCommand*, e em resposta ao item escolhido "Abrir", é invocada a função *OpenFile* pertencente ao objecto grelha. A função *OpenFile* executada é a função definida na classe *BCFileData* do Win++, em virtude de não haver redefinição desta função em *MatrixCircuit*. Por sua vez, esta função invoca a função *Clear*: por força dos mecanismos de ligação dinâmica do C++, vai ser executada a função *Clear* redefinida na classe *MatrixCircuit*.

Após a execução de *Clear*, é criada a caixa de diálogo habitual para a abertura de ficheiros (Apêndice A, ponto A.3.6.5) e é aberto o ficheiro escolhido pelo utilizador na caixa de diálogo. A próxima acção da função *OpenFile* é a chamada da função *Load* que, pelas mesmas razões já referidas para a função *Clear*, também diz respeito à classe *MatrixCircuit*. Seguidamente, na função *Load*, invoca-se a função *LoadCircuit* que realiza o interpretador para a

reconstrução do programa *Ladder*. De notar que todo o código para *BCFileData* já se encontra definido no Win ++ e que, para se ter acesso a estas facilidades, bastou derivar a classe *MatrixCircuit* de *BCFileData* e redefinir as funções *Clear* e *Load* (que são invocados internamente pelo Win++).

O interpretador é constituído pelos três módulos apresentados na figura 16: o analisador de léxico, o analisador sintáctico e o bloco de acções necessárias à correcta criação e inserção dos objectos *ladder* na grelha.

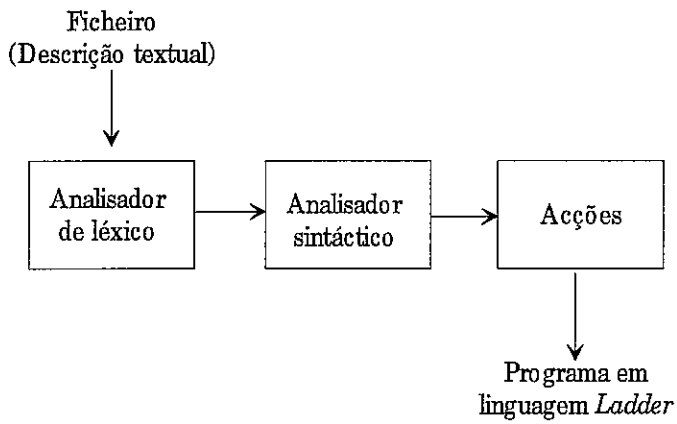


Fig. 16 - Interpretador do programa *Ladder*

Para a implementação do analisador de léxico foi utilizado o programa *lex* [21], e para a implementação do analisador sintáctico foi utilizado o programa *yacc* [22].

O programa *lex* aceita como entrada uma tabela de *patterns* e produz um analisador de léxico, sob a forma de uma função em linguagem C, capaz de reconhecer palavras que satisfazem esses *patterns*. Essa função lê um conjunto de caracteres e agrupa-os em *tokens*. O processo usado para implementar o analisador de léxico consiste em definir convenientemente o ficheiro de entrada do *lex* e, em seguida, executar o programa *lex* sobre esse ficheiro. Parte do ficheiro de entrada do *lex* é apresentado a seguir.

```
%%
const  [0-9]+
ident  [a-zA-Z][a-zA-Z_0-9]*
space  [ \t\n]+
other  .
%%
"IO"   return IO;          /* reconhece IO, que é o nome atribuído aos símbolos do tipo Contacto Normalmente Aberto
*/
...
"UDC"    return UDC;       /* reconhece UDC - símbolos do tipo Contador Ascendente/Descendente */
{space}        ;
{const}        {yylval.y_k = atoi(yytext); return Constant;}  /*  valor numérico - pode ser o parâmetro Valor ou o */
                                                              /*  número de linhas vazias */
{ident}        {strcpy(yylval.y_str, yytext); return Label;}      /* identifica uma etiqueta    */
{other}        return (yytext[0]);
%%
```

O programa *yacc* é usado para codificar a gramática de uma linguagem e constrói um *parser*. O *parser* examina os *tokens* de entrada e agrupa-os em unidades sintácticas. Nas diversas fases desse agrupamento pode haver processamento adequado utilizando rotinas em linguagem C.

O ficheiro de entrada do *yacc* apresentado a seguir descreve uma linguagem que identifica o formato do ficheiro de arquivo, dos programas em linguagem *ladder*. O resultado da execução do *yacc* sobre este ficheiro de entrada é uma função em linguagem C que é usada para reconstruir o programa no editor de programação.

A estrutura de dados de apoio ao *parser* permite armazenar informação captada por este durante o processo de descodificação de um símbolo *ladder*. Essa informação servirá para invocar convenientemente a função membro *CreatElement* do objecto do tipo grelha, descrito atrás. Sob a forma de comentário está indicado o que cada campo da estrutura representa.

```
typedef struct {
    int lin, col;          // posição para reconstrução na grelha
    int pvalue;            // valor
    int cross;             // existência de ligação vertical
    char **label;          // 1ª etiqueta, 2ª etiqueta e 3ª etiqueta
    symbol name;           // tipo de símbolo a criar para inserção na grelha
} my_yacc;

my_yacc C20Prs;            // variável global do parser
```

O *parser*, além de reconstruir o programa, detecta também possíveis erros na descrição dos ficheiros. Alguns desses erros resultam, por exemplo, de instruções onde os parâmetros da instrução estão confinados a certas áreas de memória ou de *tokens* que não identificam nenhum dos símbolos permitidos na linguagem.

As regras da gramática para descodificação do formato do ficheiro de arquivo, bem como as acções necessárias à reconstrução do programa, são apresentadas a seguir. De modo a simplificar a apresentação do ficheiro, não se mostra o código correspondente à detecção de erros.

```
circuit     :  statements          /* símbolo objectivo */
            ;
statements  :  statement
            |  statements statement
            ;
statement   :  '{'expression '}'   {  executeWindows();  /*cria um novo símbolo no editor de programação */
                                      c20Prs.lin++;       /* próxima linha */
                                      c20Prs.col = 0;     /* primeira coluna */ }
            |  '{' '/' Constant '}' {
                  c20Prs.lin += $3;    /* incrementa o número de linhas com o número de linhas vazias */
               }
            ;
expression  :  symbol
            |  expression separator symbol
            ;
symbol      :                 /* célula vazia */
            |  cross
            |  name totallabel
```

```
            |  name totallabel cross
            |  name totallabel value
            |  name totallabel value cross      /* configurações possíveis para ser símbolo */
            ;
totallabel  :
            |  c20Label              {  strcpy(c20Prs.label[0], $1); }   /* uma só etiqueta */
            |  c20Label c20Label     {  strcpy(c20Prs.label[0], $1);     /* duas etiquetas */
                                        strcpy(c20Prs.label[1], $2); }
            |  c20Label c20Label c20Label   {  strcpy(c20Prs.label[0], $1);     /* três etiquetas */
                                               strcpy(c20Prs.label[1], $2);
                                               strcpy(c20Prs.label[2], $3); }
        ;
name        :  IO     { c20Prs.name = C_IO; }       /* Contacto Normalmente Aberto */
            |  IC     { c20Prs.name = C_IC; }       /* Contacto Normalmente Fechado */
               ...
            |  UDC  { c20Prs.name = C_UDC; } /* Contador Ascendente/Descendente */
            ;
/* quando se encontra um separador pode-se passar para a célula seguinte da grelha, e pode-se criar o objecto ladder
da célula anterior */
separator   :  ','  {  executeWindows();
                       c20Prs.col++;        /* próxima coluna */     }
            ;
cross       :  '!'  {  c20Prs.cross = TRUE;   }  /* existência de uma linha vertical */
            ;
value       :  '(' '#' Constant ')'  {  c20Prs.pvalue = $3;  }  /* a constante corresponde ao valor */
            ;
c20Label    :  Label  {  strcpy($$, $1);   }      /* detecção de uma etiqueta */
            ;
```

A função *loadCircuit* tem por função inicializar a estrutura de dados de apoio ao *parser* e invocar a função *yyparse* que realiza o *parser.*

```
int loadCircuit(BCFile *fp, MatrixCircuit *circ)
{
/* incialização das variáveis usadas no parser */
    c20Prs.lin = c20Prs.col = c20Prs.pvalue = 0;  /*inicializa a linha, a coluna e o valor */
...
    return yyparse();      /* invoca o parser que reconstrói o programa */
}
```

A função *executeWindows* é invocada internamente pelo *parser* quando é detectado um símbolo. Esta função cria o símbolo *ladder* e insere-o na grelha, insere (caso exista) a ligação vertical e reinicializa algumas variáveis do *parser.*

```
void executeWindows()
{
    /*  cria o objecto e insere-o na grelha -winCircuit é um ponteiro para o objecto grelha */
    winCircuit->CreateElement( c20Prs.col, c20Prs.lin, c20Prs.name, c20Prs.label, c20Prs.pvalue);
    if(c20Prs.cross)
        winCircuit->InsertLineV(c20Prs.col, c20Prs.lin); /* cria uma ligação vertical */
    /* inicializa as variáveis do parser */
    c20Prs.name = C_NONE;
    ...
}
```

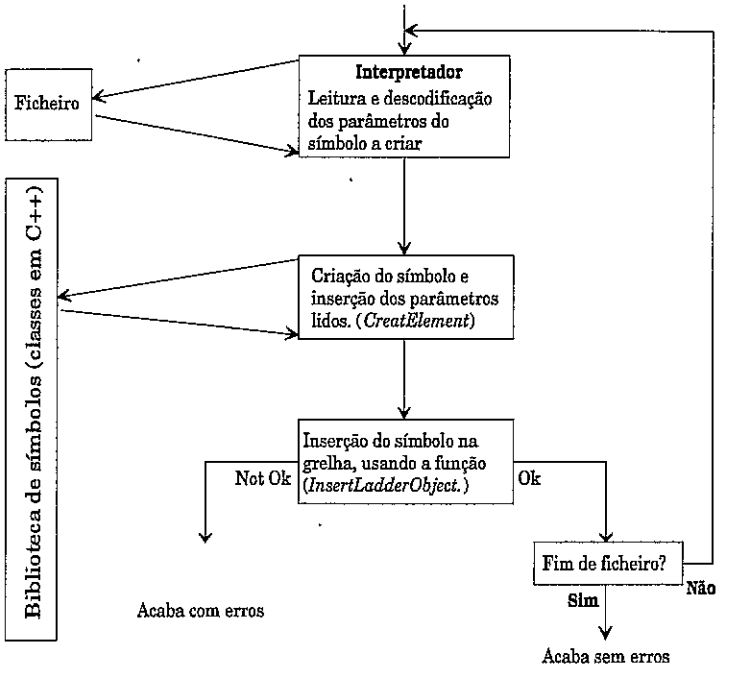


Fig. 17 - Reconstrução de um programa *ladder* a partir do ficheiro de arquivo

A sequência de acções para a reconstrução do programa é mostrada na figura 17. O interpretador, realizado pelo *parser*, tem por função extrair o símbolo (e os seus parâmetros), do arquivo. A função *CreatElement* cria o símbolo com os parâmetros e insere-o na grelha, invocando, para isso, a função membro *InsertLadderObject.*

5.2. Editor de estímulos

Um autómato programável, em função da leitura do estado das suas entradas e de acordo com o programa de controlo residente em memória, comanda as suas saídas. Existe então a necessidade de introduzir as entradas para o PLC virtual e de verificar o comportamento das saídas, por forma a avaliar o funcionamento do programa.

Na sua maioria os PLCs utilizam quase exclusivamente variáveis do tipo *bit*, pois a maior parte dos sinais a processar, em aplicações envolvendo PLCs, evolui no tempo, mas assumindo apenas dois estados possíveis.

Perante estes argumentos afigura-se interessante a ideia (não disponível em *software* comercial de PLCs) de utilizar diagramas temporais para estimular as entradas do PLC virtual, já que desse modo é possível variar o seu estado em instantes bem definidos no tempo. Assim, foi tomada a decisão de criar um editor, com características idênticas às do editor de programação, e que permitisse definir os diagramas temporais dos estímulos das entradas em uso no programa *ladder*. Além disso, este editor deveria também possibilitar a análise dos dados resultantes da simulação do programa, de modo a que o utilizador pudesse visualizar as entradas, saídas ou pontos internos do PLC virtual, sincronamente no tempo.

Do ponto de vista do utilizador, o funcionamento do editor de estímulos encontra-se descrito pode ver-se em apêndice (Apêndice A, capítulo A.4).

5.2.1. Definição de um estímulo

No contexto deste trabalho, designa-se por estímulo uma entidade que suporta dois tipos distintos de informação: um endereço e um diagrama temporal. O endereço identifica a localização do *bit* na memória do PLC virtual, e o diagrama temporal mostra o comportamento do estado do *bit* ao longo do tempo, neste caso ao longo do tempo de simulação. Este endereço referencia um *bit* que pode corresponder a uma entrada ou saída física ou a um ponto interno na memória.

Para caracterizar esta entidade (estímulo) foi criada uma classe, da qual se apresentam a seguir alguns dados e funções membro. Objectos instanciados desta classe representam estímulos, suportando o seu armazenamento em memória e permitindo a sua modificação e visualização.

```
Class Stimulus:public BCObject
{
protected:
    ...
    WORD Order;                     // ordem numa lista de estímulos
    StimulusEditor *opStimulusEditor;  // ponteiro para a janela de display do diagrama temporal
    StimulLabelEdit *labelStimul;   // ponteiro para o objecto de display do endereço
    BCRectangle oRect;              // rectângulo (em pixeis) ocupado pelo diagrama temporal do estímulo
    CHAR *StimulusLabel;            // suporte em memória para o endereço do estímulo (etiqueta)
    CHAR *StimulusBits;             // suporte em memória para o diagrama temporal do estímulo
    WORD numberOfPoints;            // número de pontos do diagrama temporal
    BCSize StimulusSizeInPixel;     // tamanho, em pixeis, de cada ponto do diagrama temporal
public:
    ...
    VOID DrawBit(WORD opPos);       // desenha o estado de um ponto do diagrama temporal
    VOID Draw(VOID);                // desenha o diagrama temporal na janela de display
    VOID SetStimulusLabel(CHAR *);  // modifica o endereço do estímulo
    VOID SetStimulusBits(CHAR *);   // modifica o diagrama temporal dos estímulos
    VOID ChangeBit(WORD oPonto, CHAR opValue); // modifica o estado de um ponto no diagrama temporal
    VOID ResetStimul(VOID);                    // inicializa o estímulo
    VOID SetNumberOfPoints (WORD nP){numberOfPoints = nP;}  // modifica o número de pontos do estímulo
    BOOL IsThisStimulusInPos(BCPosition oPos);  // verifica se a posição oPos pertence ao rectângulo do estímulo
    // modifica o estado do ponto correspondente à posição oPos. Acima da linha média do diagrama o estado é ON
    VOID ModifyValueAtPos(BCPosition oPos);
    ...
};
```

5.2.2. As janelas do editor de estímulos

O editor de estímulos é composto por várias janelas: uma janela *popup* que contém os menus e as opções, e que gere as comunicações entre as outras janelas, uma janela para edição dos endereços (caixa de diálogo), uma janela de informações e uma janela para desenho e *display* dos diagramas temporais. Uma das razões que levaram à criação de várias janelas prende-se com o facto de haver vários tipos de informação, procurando-se, por isso, que cada tipo de informação fosse visualizado em janela própria.

A janela dos diagramas temporais

Esta janela permite visualizar e introduzir os diagramas temporais. Para isso, ela não tem mais do que gerir uma determinada quantidade de objectos do tipo *Stimulus*, fornecendo o suporte para o *display* e modificação dos mesmos. Por ser uma janela, derivou-se uma classe de *BCChildWindow* para a representar e consequentemente, foram redefinidas algumas funções virtuais de suporte a eventos.

```
class SimulationEditor : public BCChildWindow
{
    SimulationWindow *opSimulationWindow;      // ponteiro para a janela mãe
    Stimulus **InputOutput;                    // conjunto de estímulos suportado
    WORD numberOfStimulus;                     // número de estímulos
    BCCursor *lineCur;                         // cursor do tipo linha para melhor análise temporal
    BCCursor *modifyCur;                       // cursor para modificação dos diagramas temporais
    BCCursor *clearCur;                        // cursor para apagar estímulos
    WORD numberOfPoints;                       // número de pontos dos estímulos
    WORD SizeOfPointInPixel;                   // tamanho em pixeis de cada ponto (para ampliação)
    ...
protected:
    // redefinição de funções pertencentes a classes acima na hierarquia
    VOID Paint(BCEvent *);
    VOID MouseDown(BCEvent *);
    VOID MouseUp(BCEvent *);
    VOID MouseDblClk(BCEvent *);
    VOID MouseMove(BCEvent *);
    VOID MouseDrag(BCEvent *);
    VOID Activation(BCEvent *);
    VOID VertScroll(BCEvent *);
    VOID HorizScroll(BCEvent *);
    VOID KeyDown(BCEvent *);       // para atender eventos de tecla premida
    VOID KeyUp(BCEvent *);         // para atender eventos de tecla levantada
    WORD Initialize(VOID);
public:
    VOID SetModifyCur (VOID);          // instala o cursor de modificação
    Stimulus *GetFirstVisible(VOID);   // retorna o primeiro estímulo visível
    // devolve o estímulo, o número do ponto, e o estado do ponto correspondente à posição oPos (pixel)
    Stimulus *GetPosValue(BCPosition oPos, WORD &oPonto, CHAR &oEstado);
};
```

A função *MouseDown* contém código para editar ou apagar estímulos.

```
VOID SimulationEditor:: MouseDown(BCEvent *opEvt)
{
BCPosition oPos = opEvt->GetPos();                  // oPos = posição do rato
    ...
    for(int i = 0; i< numberOfStimulus; i++)
        if (InputOutput[i]->IsThisStimulusInPos(oPos) ) {   // verifica qual o estímulo mostrado sobre a posição oPos
            if (mouseAction == NORMAL_ACTION)               // acção normal de edição do diagrama temporal
                InputOutput[i]->ModifyValueAtPos(oPos);     // modifica o estado do ponto de acordo com oPos
            if (mouseAction == CLEAR_ACTION )               // acção para apagar o estímulo
                InputOutput[i]->ResetStimul();              // inicializa o estímulo
            break;                                          // termina, pois já encontrou o estímulo
        }
}
```

Por exemplo, a função *Paint*, entre outras acções, limpa a janela e faz o *display* de todos os estímulos da janela.

```
VOID SimulationEditor :: Paint(BCEvent *)
{
    Clear();        // limpa a janela
    for (int i = 0 ; i<numberOfStimulus; i++ )
        InputOutput[i]->Draw();     // desenha o estímulo que ocupa o número de ordem i
    ...
}
```

A janela de endereços

Esta janela permite a edição e a visualização dos endereços dos estímulos. Definiu-se esta janela como uma caixa de diálogo, pois a gestão dos objectos aí presentes (a maior parte da gestão é feita internamente pelo Windows) torna-se mais simples, e além disso pode-se utilizar o utilitário Workshop para definir a sua aparência e também os controlos que possui. Esta janela mantém objectos instanciados duma classe *StimulLabelEdit*, que é derivada da classe base do Win++ para representar controlos de edição de palavras, *BCStringEdit*. Esses objectos (controlos) vão ser usados para editar e visualisar os endereços dos estímulos.

```
class EditStimulusDialog : public BCModelessDialog
{
    SimulationWindow *opSimulationWindow;       // ponteiro para a janela que gere o editor
    StimulLabelEdit **labelStimul;              // suporte para os controlos criados na caixa de diálogo
    Stimulus **opStimulus;                      // ponteiros para os estímulos
protected:
    WORD Initialize(VOID);              // inicializa, entre outros, os controlos para edição dos endereços
    VOID EditChange(BCEvent *opEvt); // invocada quando se modifica alguma palavra suportada pelos controlos
public:
    EditStimulusDialog (Stimulus **opStim, SimulationWindow *opParent);  // construtor da classe
    ...
};
```

Para permitir a modificação, pelo utilizador, dos endereços nos estímulos redefiniu-se a função virtual *EditChange*, que é invocada internamente pelo Win++ sempre que uma palavra de qualquer controlo presente na caixa de diálogo é modificada. Nessa função detecta-se o controlo modificado, acede-se à palavra modificada e, em seguida, altera-se o endereço do estímulo correspondente, invocando a função membro adequada da classe *Stimulus*.

```
VOID EditStimulusDialog :: EditChange(BCEvent *opEvt)
{

    CHAR caText[MAX_LABEL_SIZE];
    // descobre o número de ordem do controlo cuja palavra foi modificada, através de informação retirada do evento
    WORD oOrd = opEvt->GetControlID() - IDC_EDIT;
    labelStimul[oOrd]->GetText(caText, MAX_LABEL_SIZE-1); // carrega em caText a palavra do controlo
    opStimulus[oOrd]->SetStimulusLabel(caText);                    // modifica o endereço do estímulo correspondente
    ...
}
```

Estes controlos (objectos do tipo *StimulLabelEdit*) contêm mecanismos para *display* de palavras e para o controlo da entrada de caracteres; por isso, para carregar uma nova palavra no controlo (por exemplo numa acção de carregar estímulos pré-gravados no disco), só é necessário invocar a função membro *SetText* com o parâmetro adequado.

A janela de gestão

A janela de gestão do editor é a janela mãe de todas as janelas que constituem o editor de estímulos. Além de suportar as outras janelas, esta janela também mantém os parâmetros de simulação (Apêndice A, ponto A.4.4.1), permitindo visualizá-los (através da janela de informações) e modificá-los (através das opções e/ou menus).

O objecto representativo desta janela é instanciado da classe *SimulationWindow*, derivada de *BCPopUpWindow*, a qual constitui a classe base para representar janelas tipo *popup*.

```
class SimulationWindow : public BCPopUpWindow
{
    C20FastSimul *opFastSimul;                    // objecto PLC virtual - simulação rápida
    C20RealSimulNormal *opRealSimulNormal; // objecto PLC virtual - simulação em tempo real
    C20RealSimulInteractive *opRealSimulInteractive;  // ponteiro para a janela interactiva
    EditStimulusDialog *opEditStimulusDialog;          // ponteiro para a janela de endereços
    SimulationEditor *sEditor;                         // ponteiro para a janela dos diagramas temporais
    SimulationStatusWindow *opStatus;                  // ponteiro para a janela de informações
    SimulatorC20Window *opParent;                      // ponteiro para a janela de aplicação
    BCBitmap *opBitmapSave,...*opBitmapZoom;           // ponteiros para os bitmaps das opções
    double simulationTime;              // tempo de simulação
    double stepPeriod;                  // periodo de varrimento do programa durante a simulação
    WORD numberPoints;                  // número de pontos dos estímulos
    WORD zoomSizePixel;                 // número de pixeis por ponto do estímulo
    FileStimulus *opFileStimulus;       // objecto que representa o ficheiro de arquivos dos estímulos
    BCMenu *opSimMenu;                  // ponteiro para o objecto menu da janela
protected:
    WORD Initialize(VOID);
    VOID MenuCommand(BCEvent *);
    VOID ReSize(BCEvent *);
    VOID Paint(BCEvent *);
    BOOL QueryEnd(BCEvent *);
    VOID MenuSelect(BCEvent *);
    VOID Activation(BCEvent *);
public:
    // carrega os parâmetros de uma simulação - invocada na acção de leitura de estímulos pré-gravados em disco
    VOID SetSimulationParameters(double timeSimul, double stepPer, WORD zoomPixel);
    VOID InitSimulationValues(VOID);  // inicializa os parâmetros de simulação com valores por defeito
    VOID StopSimulation(VOID);        // pára a simulação em tempo real
};
```

A figura 18 exemplifica os sentidos da comunicação entre alguns dos objectos constituintes do editor de estímulos (janelas e estímulos). A janela do editor de estímulos é a janela mãe da janela de endereços e da janela dos diagramas temporais, e como tal, tem a capacidade de gerir as duas, e ainda de servir como suporte para a comunicação entre ambas. Cada janela (endereços e diagramas temporais) permite editar e visualizar cada uma das partes distintas dos estímulos, como são, os endereços e os diagramas temporais.

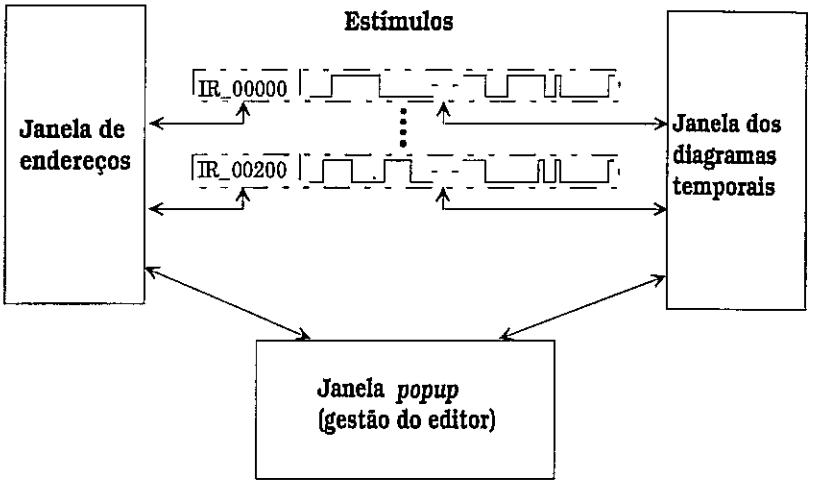


Fig. 18 - Comunicação entre as janelas para edição dos estímulos

5.3. Editor da simulação interactiva

A ideia da simulação interactiva surge duma análise de algumas famílias de autómatos que permitem a interligação de módulos para simular as entradas através de interruptores, possibilitando a estimulação dos autómatos sem qualquer ligação a entradas reais. Isto é particularmente vantajoso na fase de desenvolvimento dos programas, principalmente na implementação de controlo lógico binário, tornando muito fácil o teste de situações variadas.

A ideia do utilizador actuar sobre os interruptores para modificar o estado das entradas do autómato real é implementada no sistema em análise com o premir do botão do rato sobre um objecto do tipo entrada.

Mais uma vez se remete o leitor para o Apêndice A, ponto A.5.4, para verificar o funcionamento do editor, o qual do ponto de vista de programação é um objecto do tipo *C20RealSimulInteractive.*

```
class C20RealSimulInteractive :  public C20RealSimul
{
    WORD editOrSimul;          // sinaliza a permissão de edição da janela ou a permissão de simulação
    InputOutputRadioButton *opInputs[ MAX_INT_SIMUL_INPUTS];          // objectos tipo entrada
    InputOutputRadioButton *opOutputs[ MAX_INT_SIMUL_OUTPUTS];        // objectos tipo saída / ponto interno
    BCBitmap *opBitmapIntLoad,...*opBitmapIntStop;                    // bitmaps das opções
    WORD numberInputs;                  // número de objectos do tipo entrada
    WORD numberOutputs;                 // número de objectos do tipo saída
    InteractiveSimulFile *opIntSimulFile;   // ponteiro para o objecto de armazenamento em ficheiro
    BCMenu *opSimIntMenu;               // ponteiro para o menu
    BCStaticText *opStaticTextInfo;     // para display de informação na janela
protected:
    VOID MenuCommand(BCEvent *);
    WORD Initialize(VOID);
    VOID ButtonSnglClk(BCEvent *);
    VOID ButtonDblClk(BCEvent *);
    VOID Paint(BCEvent *);
    VOID MouseDown(BCEvent *);
    VOID MouseUp(BCEvent *);
    VOID MouseDrag(BCEvent *);
```

```
    VOID Activation(BCEvent *);
    VOID TimerEvent(BCEvent *);          // para atender eventos do temporizador da simulação
    VOID ReSize(BCEvent *);
    BOOL QueryEnd(BCEvent *);
public:
    ...
    InputOutputRadioButton *GetButtonAtMousePos(BCPosition oPos);      // devolve o objecto na posição oPos
    BOOL InitSimulation();               // inicia a simulação interactiva
    VOID StopSimulation(VOID);           // pára a simulação interactiva
    VOID UpdateInputStimulus();          // lê o o estado actual dos objectos de entrada -antes da simulação
    VOID UpdateOutputStimulus();         // actualiza o estado dos objectos de saída -após a simulação
};
```

Por exemplo, para actualizar o estado das entradas só é necessário atender o evento de *ButtonSnglClk*, que sinaliza que o estado do botão foi mudado por acção do rato ou teclado. Nesta função, detecta-se o botão cujo estado mudou e actualiza-se o objecto tipo entrada, de modo a que, no próximo varrimento da simulação, o estado da respectiva entrada já esteja actualizado. Caso a opção activa seja a Borracha (Apêndice A, ponto A.3.3) deve-se apagar o objecto da janela.

```
VOID C20RealSimulInteractive:: ButtonSnglClk(BCEvent *opEvt)
{
    InputOutputRadioButton *opIn = (InputOutputRadioButton*)opEvt->GetControl();  // objecto cujo estado variou
    ...
    if (opIn->IsInput() && editOrSimul == SIMUL_MODE )      // o objecto é uma entrada
        opIn->SetState(!opIn->GetState());                  // muda o estado do objecto
    else
        if (editOrSimul == CLEAR_MODE ) {       // para apagar o objecto entrada ou saída
            opIn->SetActive(FALSE);             // inactiva o objecto
            opIn->SetLabel("");                 // limpa a etiqueta do objecto
            opIn->SetAddress("");               // limpa o endereço do objecto
            opIn->Hide();                       // esconde o objecto tornando-o invisível
        }
}
```

Capítulo 6
Virtualização do PLC

"Virtualização de" transmite a ideia de emular, ou mais simplesmente de imitar uma realidade. O aparecimento dos computadores veio fornecer um suporte para imitar essa realidade de uma forma mais sistemática, tendo-se tornado a simulação por computador uma ferramenta de engenharia bastante importante, quer para o desenvolvimento de novos produtos, quer para a pesquisa de novas soluções. De facto, e segundo BIEN [23], a simulação por computador permite fazer análise de *design*, programação *off-line* e treino.

Também na área do ensino, a simulação tem vindo a assumir um papel importante em matérias onde é necessário dispender recursos de elevado custo. Por exemplo, e citando WHITE [8] e RAZ [24], foram realizados alguns trabalhos no intuito de auxiliar o ensino de controlo de robôs em que a ideia geral é a simulação do comportamento do robô.

No caso concreto do presente trabalho, como já foi referido, o objectivo é simular o funcionamento de um autómato programável de modo a que o sistema resultante possa vir a ser utilizado no ensino de programação de autómatos programáveis, e ainda ajudar no teste rápido de soluções para um determinado problema de controlo.

6.1. Modelo para a arquitectura do PLC virtual

A arquitectura dos autómatos programáveis já foi descrita em capítulos anteriores. Duma forma simplista, um autómato é um sistema computacional com memória que lê as suas entradas, e comanda as suas saídas de acordo com as entradas e o programa de controlo armazenado em memória.

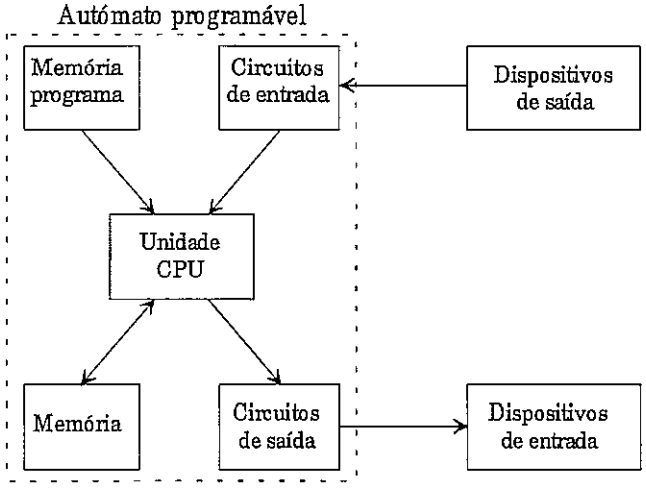


Fig. 19 - Arquitectura funcional de um PLC (sem fonte de alimentação)

No presente trabalho, o modelo para a virtualização de um autómato programável encontra-se representado na figura 20 e traduz-se pela substituição:

- do CPU por um simulador orientado por objectos;
- do programa em memória por um editor de programação;
- da memória do PLC por uma área de memória do PC;
- das entradas reais por acções introduzidas pelo rato ou pelo teclado;
- das saídas por uma sua imagem visualizada no monitor.

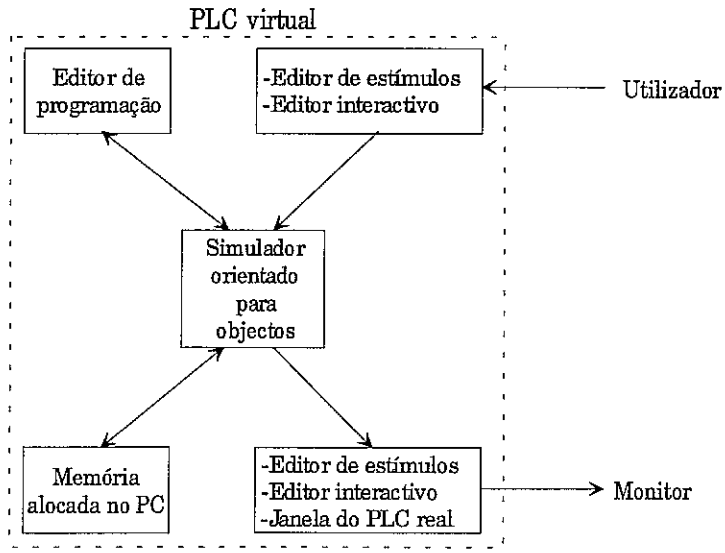


Fig. 20 - Modelo para a arquitectura do PLC virtual

6.2. Linguagem do PLC virtual

Para o editor de programação havia que escolher uma das linguagens existentes para programação de autómatos.

Como os diagramas de escada constituiam uma técnica muito usada para especificar sistemas de controlo, a linguagem *ladder* surgiu naturalmente e é ainda hoje a linguagem mais utilizada para a programação de PLCs. Na bibliografia podem facilmente encontrar-se opiniões que revelam a importância da linguagem *ladder* na programação de PLCs: os PLCs são tipicamente programados através da linguagem *ladder* [25]; a programação de PLCs é fácil pois ela é geralmente no formato *ladder* [26]; A lógica em formato *ladder* é uma forma de inteligência artificial. Não é o *state-of-art* que conta na engenharia de controlo, mas sim o nível de aceitação [27]. Pelas razões apontadas e porque quase invariavelmente os fabricantes fornecem a linguagem *ladder* para os seus autómatos, foi esta a linguagem escolhida para a programação do PLC virtual.

Esta forma gráfica de programação pode ser considerada uma forma de programação visual (qualquer sistema que permite a especificação de um programa de uma forma bidimensional ou a mais dimensões). É um dado adquirido que um estilo de programação mais visual pode ser mais facilmente apreendido, principalmente por programadores pouco

experimentados; de facto, alguns sistemas de programação visual têm demonstrado com sucesso que não programadores podem produzir programas complexos com pouco treino [28].

6.3. Memória do PLC virtual

A memória do PLC virtual espelha a memória do PLC real que está a ser emulado. Como vimos no ponto 2.4.2, a memória do autómato C20H está dividida em áreas, sendo cada uma identificada por uma designação.

Na definição da classe base do PLC virtual, essa memória existe integralmente no objecto PLC virtual. A razão do *static* prende-se com a existência da vários objectos do tipo PLC virtual no sistema, os quais partilham a memória do PLC virtual base.

```
class C20Omron : public BCPopUpWindow
{
 protected:
    // dados estáticos do PLC virtual, são os mesmos para todas as instâncias de PLCs
    // memória do PLC virtual
    static WORD IR[INTERNAL_RELAY_MEMORY+1];          // memória Internal Relay
    static WORD SR[SPECIAL_RELAY_MEMORY+1];           // memória Special Relay
    static WORD AR[AUXILIARY_RELAY_MEMORY+1];         // memória Auxiliary Relay
    static WORD DM[DATA_MEMORY+1];                    // memória Data Memory
    static WORD HR[HOLDING_RELAY_MEMORY+1];           // memória Holding Relay
    static WORD TC[TIMER_COUNTER_MEMORY+1];           // memória Timer/Counter
    static WORD LR[LINK_RELAY_MEMORY+1];              // memória Link Relay
    static WORD TR;                                   // memória Temporary Relay - só acedida por bit
    // outros dados
    static SimulC20Window *opParent;                  // ponteiro para a janela de aplicação
    static SimulationWindow *opSimulationWindow;      // ponteiro para o editor de estímulos
    static MatrixCircuit *Circ;                       // ponteiro para a grelha que contém o programa
    static BaseShape *rightObjects[NUMBER_VERTICAL_CELL];  // objectos mais à direita de cada linha
    // parâmetros de configuração da simulação
    static WORD numberOfSteps;                        // número de pontos a simular
    static double stepPeriod;                         // periodo de varrimento - scan time
    static WORD numberClocksPerScan;                  // nº de clocks por cada scan time
    static WORD stepInSimulation;                     // nº do ponto corrente a ser simulado
// informação sobre cada estímulo - diagrama temporal, endereço do bit em memória (palavra + máscara de acesso)
    InputStim inputStimul[MAX_C20_INPUTS];            // estímulos das entradas
    OutputStim outputStimul[NUMBER_OF_STIMULUS];// estímulos para guardar os resultados da simulação
    ...
    public:
    WORD *GetIR(WORD index);     // devolve um ponteiro para a palavra em index, na área de memória IR
    VOID SetIR(WORD index,WORD value){ IR[index]=value;}  // modifica a palavra index na área IR
    // para desenhar as mudanças da janela do PLC virtual no fim de cada varrimento
    virtual BOOL VOID InitSimulation(VOID);           // inicializa para a simulação
    virtual BOOL SimulationC20(VOID);          // executa um varrimento no programa - simula a sua execução
    virtual VOID UpdateOutputStimulus(WORD oStepTime);  // faz o update dos estímulos de saída
    virtual VOID UpdateInputStimulus(WORD oStepTime);   // inicializa as entradas para novo varrimento
    virtual VOID StopSimulation(VOID){}               // para parar a simulação
    ...
};
```

A classe base do PLC virtual é derivada de uma classe tipo janela (*BCPopUpWindow*). Isto deve-se a duas razões, uma que se prende com a existência de uma janela de *display* do próprio PLC, justificada pela necessidade de programar um temporizador do Windows (em alguns modos de simulação) e de receber os seus eventos no PLC virtual (os eventos do temporizador são enviados para um objecto do tipo janela).

A figura 21 representa a entidade PLC virtual, através dos seus componentes CPU e memória, bem como a sua funcionalidade, isto é, a interligação entre os vários componentes. Esta figura elucida o comportamento de objectos instanciados de classes, cuja classe base é a classe *C20Omron*. A função do CPU principal será analisada em altura própria, podendo-se já referir que este serve de excitador do processo de simulação.

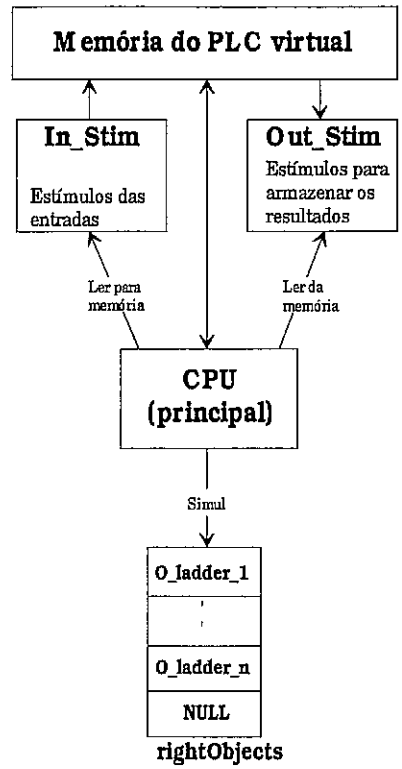


Fig. 21 - Interligação entre os vários componentes do objecto *C20Omron*

O endereçamento de memória do PLC virtual é idêntico ao endereçamento de memória do autómato real que está a ser emulado (por razões de familiarização com os endereços físicos do PLC real). Para endereçar um *bit* na memória é necessário especificar a área de memória (IR, AR, TC, ...), o número da palavra (de 16 *bits*) e o número do *bit* (1 a 16) na palavra.

Cada objecto *ladder* define um conjunto de etiquetas (ver classe *BaseShape*, ponto 5.1.1), que suportam a identificação dos endereços das posições de memória, sobre as quais a instrução *ladder* correspondente opera. Além dessas etiquetas, cada objecto *ladder* suporta também um conjunto de ponteiros para a memória do PLC virtual, juntamente com um conjunto de máscaras para aceder aos *bits* nas palavras referenciadas por esses ponteiros (figura 22).

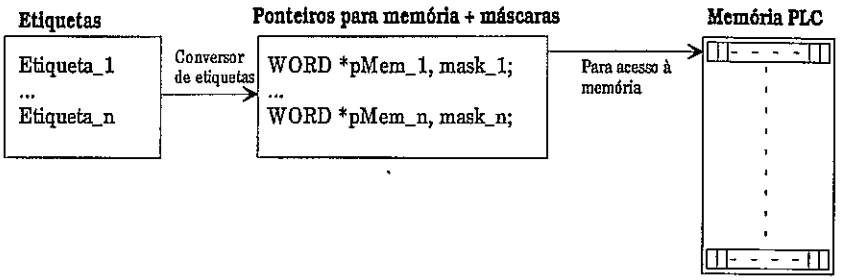


Fig. 22 - Suporte para um objecto *ladder* operar sobre a memória

Como vimos, a questão de endereçamento é bastante simples. Existem, no entanto, questões respeitantes às áreas de memória que cada instrução pode usar no PLC real, que têm que ser garantidas também no sistema PLC virtual. Para isso foi instalado um *switch* (reconstrução ou detecção de erros nos endereços das instruções *ladder*) no *parser* de reconstrução do programa, o qual permite correr o código que garante a concordância dos endereços das intruções do PLC virtual com as instruções do PLC real.

6.4. Simulador

O modelo para a arquitectura do PLC virtual inclui um simulador que emula o comportamento do autómato real. Para tal o simulador deverá compreender a linguagem *ladder* e executar sobre a memória do PLC virtual todas as instruções implementadas.

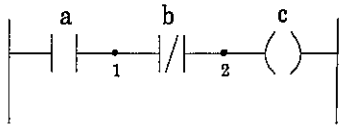
Um dos objectivos principais, para o desenvolvimento do simulador, é que este permaneça inalterável com a introdução de novas instruções ou com a criação de novos PLCs virtuais. Esta ideia implica que o código já realizado para a simulação não seja modificado, e também que a estrutura da simulação não seja alterada com futuros *upgrades*.

No que diz respeito à implementação de novas instruções, a estratégia seria o desenvolvimento de um algoritmo genérico (independente do tipo de objectos *ladder* a simular), e distribuído (i. é., com cada nova instrução a implementar a sua própria simulação, e não o simulador principal do PLC virtual a implementar a simulação da nova instrução).

A ideia de vários PLCs virtuais resulta da previsão de novos tipos de simulação a inserir no sistema PLC virtual. Isto obriga a que se implementem mecanismos que sustentem a criação de novos PLCs virtuais (para simular programas de maneira diferente da dos modos implementados neste trabalho), e permitam a sua rápida inserção no sistema. Isto poderá ser conseguido através de uma hierarquia apropriada de PLCs virtuais.

A construção de um simulador utilizando técnicas de programação por objectos, de acordo com [29], envolve a comunicação entre objectos ao longo do tempo. No caso presente existem objectos *ladder* que comunicam entre si, através de eventos de simulação, e actuam, de acordo com a instrução que implementam, sobre a memória do PLC virtual. De facto, esta forma de simulação enquadra-se no modo como é intrepretado um diagrama *ladder*.

Num diagrama *ladder* uma solução possível consiste em, a partir do início de continuidade, fazer deslocar um "testemunho" de continuidade lógica ao longo dos vários degraus que constituem o programa. Vejamos o exemplo do degrau *ladder* correspondente à equação lógica $a.\overline{b}=c$.

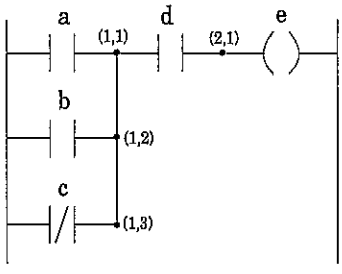


A execução da instrução, cuja etiqueta é **c**, só é realizada após a "chegada" ao nodo **2** de um sinal indicativo da existência ou não de continuidade lógica, desde a linha de início de continuidade até ao nodo em questão (comparável a um circuito eléctrico que está fechado até ao nodo **2**). Esse sinal corresponde à condição de execução da instrução **c**. Mas, para que ao nodo **2** chegue essa condição, esta já deve ter chegado ao nodo **1**, o que sugere que a simulação de um objecto *ladder* implique a existência de uma condição de execução de entrada, condição essa que resulta da auto-simulação do objecto ligado à sua esquerda. Podemos, portanto, considerar que esse esse valor de entrada (condição de execução) funciona como um testemunho que vai permitir que o objecto se auto-simule.

A condição de execução de uma instrução será TRUE quando houver continuidade lógica até ao nodo que precede o objecto *ladder*, e será FALSE se não houver continuidade lógica.

Outra questão a considerar ocorre quando um ou mais objectos *ladder* estão ligados na vertical. É necessário saber como é condicionada a continuidade lógica nestas situações.

Por exemplo, no diagrama *ladder* apresentado abaixo, a continuidade no nodo **(1,1)** depende dos três objectos (**a**, **b** e **c**). Este diagrama pode ser usado para representar a equação lógica $(a+b+\overline{c}).d=e$, o que sugere que os nodos **(1,1)**, **(1,2)** e **(1,3)** são iguais entre si e que realizam o OR lógico dos três caminhos de continuidade assegurados pelos objectos **a**, **b** e **c**.



6.4.1. Varrimento do programa *ladder*

Recordamos que um PLC é uma máquina cíclica, isto é, o programa nele contido é executado indefinidamente, sendo somente interrompido por acção do utilizador ou por falha de alimentação. Cada execução do programa é designada por varrimento.

Por analogia com o processo de execução de um programa num PLC real, o PLC virtual simula o programa de uma forma idêntica. A figura 23 é ilucidativa desse facto, vendo-se que as tarefas envolvidas na execução de um programa no PLC real foram transformadas em funções membro do PLC virtual.

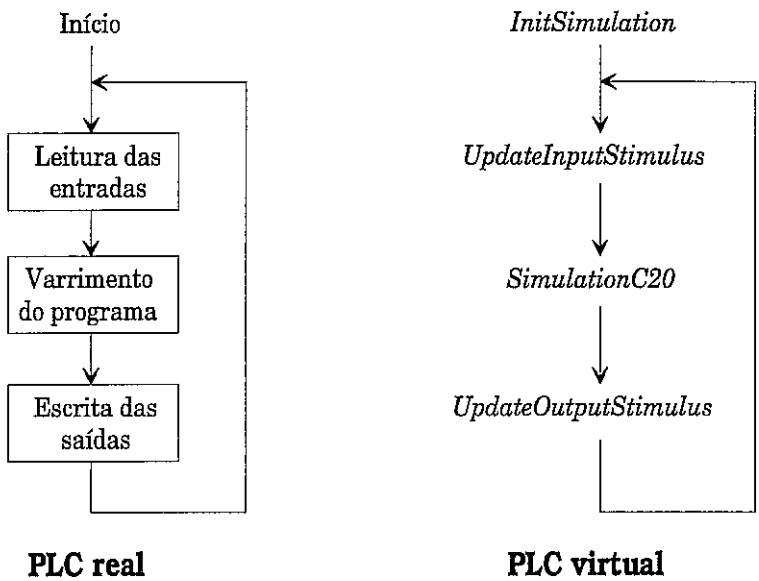


Fig. 23 - PLCs (real e virtual) em modo execução (*run mode*)

Como a simulação de um programa corresponde à simulação de uma série de varrimentos do mesmo programa, trataremos seguidamente o processo de simulação de um varrimento.

O algoritmo de simulação é baseado na passagem de informação (através de eventos) de objectos *ladder* situados à esquerda na grelha (início de continuidade) para objectos à direita na grelha. Simultaneamente, o conteúdo das posições de memória relevantes no PLC virtual é modificado de acordo com o tipo e a acção dos objectos no contexto do programa *ladder* [30].

O processo de simulação é dividido em duas fases: a inicialização e a simulação. Na fase de inicialização, a informação relevante para a simulação, nomeadamente as ligações entre os objectos *ladder* (*MakeConnections* de *MatrixCircuit*), endereços de memória e valores associados com contadores e temporizadores, são inicializados. Também nesta fase são detectados os possíveis erros nos degraus *ladder* (*VerifyCircuitToSimul* de *MatrixCircuit*), nomeadamente símbolos não ligados ou endereços não válidos. De modo a acelerar o processo de simulação é criado um *array* (*rightObjects*) que contém ponteiros para todos os objectos *ladder* (para todas as linhas do programa) situados mais à direita na grelha. A função membro *InitSimulation* é a responsável por essa inicialização. Esta função é virtual pois a implementação da inicialização nos diversos PLCs virtuais pode ser diferente, acontecendo o mesmo com o início da simulação.

O uso da matriz *rightObjects* acelera o processo de simulação porque, uma vez construída no início da simulação, evita a procura dos objectos mais à direita em todos os varrimentos do programa.

```
BOOL C20Omron :: InitSimulation()
{
    if (!Circ->VerifyCircuitToSimul())     // verifica se não existem erros nos degraus do programa ladder
        return FALSE;                      // existem erros
    // cria as ligações dos objectos ladder no programa e verifica a correcção em todas as ligações
    if (!Circ->MakeConections() )
        return FALSE;                      // existem objectos ladder que não estão ligados correctamente
    // inicia toda a memória do PLC virtual
    for (int n = 0; n <= INTERNAL_RELAY_MEMORY; n++)    // inicializa a área de memória Internal Relay
        IR[n] = 0x0000;                                 //memória Internal Relay
    ...
    // preenchimento do array dos elementos mais à direita
    BaseShape *oShape = NULL;
    for ( int i = 0, n = 0; n < NUMBER_VERTICAL_CELL; n++ ) {
        rightObjects[i] = NULL;
        col = NUMBER_HORIZONTAL_CELL -1;       // última coluna
        while( col-- >= 0 )                    // procura o elemento mais à direita na mesma linha
            if (oShape = Circ->GetXY(col,n)->GetLeftShape() ) {
                rightObjects[i++] = oShape;// encontrou - coloca um ponteiro para o elemento no array
                break;
            }
    }
    stepInSimulation = 0;          // corresponde ao primeiro varrimento do programa
    InitNewStimulus();             // inicializa os estímulos na simulação
    }
```

Por analogia com a execução de um programa num PLC real, os três módulos representados na figura 23 são emulados por três funções virtuais do PLC virtual: *SimulationC20*, *UpdateInputStimulus* e *UpdateOutputStimulus*.

A função *SimulationC20*, que representa também o CPU principal na figura 21, simula um varrimento do programa que se encontra na grelha do editor de programação. Esta começa por enviar uma mensagem de inicialização para o objecto grelha (*Circ*), de modo a inicializar todos os objectos *ladder* aí presentes para novo varrimento de simulação. Por sua vez, a grelha, por analogia com os eventos de *Draw* (ponto 5.1.3), envia para cada objecto a mesma mensagem de inicialização. Após esta inicialização, há que estimular os objectos *ladder* que se situam mais à direita no programa *ladder*, isto é, enviar-lhes a mensagem de *Simul*.

```
BOOL C20Omron :: SimulationC20()
{
    Circ->ResetForOtherSimulation(); // limpa as variáveis usadas em cada varrimento da simulação
    int n = 0;
    while ( rightObjects[n] )          // enquanto houver ponteiros para objectos ladder em rightObjects
        rightObjects[n++]->Simul();    // envia um evento de simulação para o objecto ladder rightObjects[n]
    return (TRUE);
}
```

A função *UpdateInputStimulus* lê o estado das entradas definidas nos diagramas temporais dos estímulos e actualiza as respectivas entradas do PLC virtual, o que deve ser executado antes da simulação de um novo varrimento do programa.

```
VOID C20Omron :: UpdateInputStimulus(WORD oStepTime) // actualização das entradas
{
int i=0;
// coloca as novas entradas com os estímulos correspondentes
    while (outputStimul[i].memLocation && i< MAX_C20_INPUTS) {    // o estímulo está definido
        if ( inputStimul[i].opInput[oStepTime] =='\1' )    // o estado correspondente ao varrimento actual é ON
            IR[0] |= inputStimul[i].maskBit;              // coloca a entrada i do PLC virtual no estado ON
        else
            IR[0] &= ~inputStimul[i].maskBit;     // coloca a entrada i do PLC virtual no estado ON
        i++;
    }
}
```

A função *UpdateOutputStimulus* é a definição por defeito para a função que actualiza os diagramas temporais dos estímulos após cada varrimento do programa. A actualização de um estímulo consiste em ler o estado do *bit* endereçado pelo estímulo e guardá-lo no respectivo diagrama temporal. Esta função deve ser invocada após a simulação de um varrimento, isto é, após a invocação da função *SimulationC20.*

```
VOID C20Omron :: UpdateOutputStimulus(WORD oStepTime)
{
int i=0;
    while (outputStimul[i].memLocation && i< NUMBER_OF_STIMULUS) {   // o estímulo está definido
    // Value fica com o estado do bit para o qual o estímulo foi definido - o bit pertence à memória do PLC virtual
        BOOL oValue = (*outputStimul[i].memLocation) & outputStimul[i].maskBit;
        if(oValue)
            outputStimul[i].opOutput[oStepTime] = '\1';   // guarda o estado ON, correspondente ao varrimento actual
        else
            outputStimul[i].opOutput[oStepTime] = '\0';   // guarda o estado OFF
        i++;
    }
}
```

Estas funções, *UpdateInputStimulus* e *UpdateOutputStimulus*, poderão ser redefinidas nas classes representativas de outros PLCs virtuais, sempre que as actualizações das entradas e/ou a actualização dos resultados da simulação de um varrimento se processe de maneira diferente.

6.4.2. O algoritmo de simulação

A simulação de um programa é suportada pelas funções virtuais *Simul* e *OwnerSimulation* definidas na classe abstracta *BaseShape*, mas que podem ser redefinidas nas classes representativas de objectos *ladder*. A ligação entre os símbolos gráficos (e os correspondentes objectos *ladder* que representam esses símbolos) define a forma como a informação é passada entre eles. Para cada objecto *ladder*, a função *Simul* pode ser considerada como o interpretador das ligações do objecto a outros objectos *ladder*, e a função *OwnerSimulation* executa a acção da instrução associada com o objecto *ladder*.

```
BOOL BaseShape :: Simul(WORD flagOnCall)
{
    if (flagSimulation)          // indica se o objecto actual já foi simulado, no actual varrimento,
        return (valSimulation);  // devolve o valor da simulação - condição de execução para o objecto seguinte
    BOOL valDown = FALSE, valUp = FALSE; // guardam os valores da simulação dos objectos abaixo e acima
    if (leftShape[0]) {                  // se houver um objecto ladder ligado à esquerda
        valInSimulation = leftShape[0]->Simul(LEFT_SHAPE_CALL);   // invoca a função Simul para esse objecto
        OwnerSimulation();                                        // executa OwnerSimulation do objecto ladder actual
    } else
            if( col == 0 ) {                  // à esquerda do objecto está o início de continuidade
                valInSimulation = TRUE;       // o valor de entrada para a simulação é TRUE
                OwnerSimulation();
            }
    if (orUp && flagOnCall != DOWN_SHAPE_CALL)    // existe um objecto ligado acima
         valUp = orUp->Simul(UP_SHAPE_CALL);      // invoca a função Simul para esse objecto
    if (orDown && flagOnCall != UP_SHAPE_CALL)    // existe um objecto ligado abaixo
        valDown = orDown->Simul(DOWN_SHAPE_CALL);
    valSimulation = valSimulation || valUp || valDown;   // o valor da simulação é o OR lógico dos três valores
    if(flagOnCall == LEFT_SHAPE_CALL) {    // deve-se actualizar os objectos acima e abaixo
        if(orDown)
            orDown->SetDownValSimulation(valSimulation);   // actualiza os objectos abaixo com o novo valor
        if(orUp)
            orUp->SetUpValSimulation(valSimulation);       // actualiza os objectos acima com o novo valor
    }
    return (valSimulation);      // devolve o valor da simulação
}
```

Como já foi referido, a função do CPU principal (*SimulationC20*) é ser o excitador do processo de simulação. Essa excitação traduz-se na invocação da função *Simul* para todos os objectos *ladder* existentes em *rightObjects*. Desta forma inicia-se uma navegação pela grelha, da direita para a esquerda, seguida de retorno ao objecto inicial, à semelhança do que acontece no processo de simulação implementado em [20] e [31]. Esta navegação resulta das características reentrantes da função *Simul*. Para simular a acção de um objecto, os parâmetros de entrada que lhe estão associados (condições de execução) devem ser conhecidos, o que é obtido invocando a função *Simul* dos objectos à sua esquerda. Para o objecto imediatamente à esquerda, o processo é o mesmo pelo que, assim, a grelha é percorrida desde a direita até à esquerda; a navegação para a esquerda termina quando se chega ao início de continuidade (col = 0), ou quando o objecto já foi simulado para o mesmo varrimento (flagSimulation = TRUE).

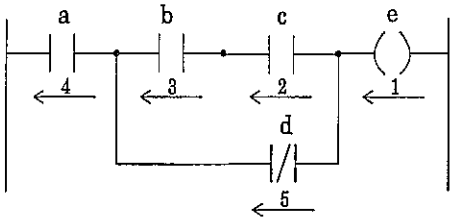
A navegação da esquerda para a direita resulta dos regressos das sucessivas invocações da função *Simul* para diferentes objectos. Após a execução da função *leftShape->Simul* para um objecto *ladder*, a sua função de *OwnerSimulation* é também executada.

A resolução das ligações verticais do objecto (que correspondem a *Or's* lógicos no contexto da linguagem *ladder*), obtém-se invocando as funções *upShape->Simul* e *downShape->Simul*. De facto, cada objecto *ladder* pertence a duas listas ligadas de objectos *ladder*, uma na horizontal e outra na vertical. A lista horizontal é ligada através de objectos *leftShape* e a lista na vertical, a existir, é uma lista duplamente ligada, pois se um objecto conhece aquele que está acima, também o objecto acima conhece o objecto que está abaixo. Esta lista é, portanto, mantida pelas ligações verticais entre objectos.

Já foi referido que a função virtual *Simul* opera como um interpretador da linguagem *ladder*. Como para todos os objectos que ocupam uma só célula (e portanto têm uma só entrada), a interpretação do contexto do programa *ladder* é idêntica, a função *Simul* definida na classe *BaseShape* serve todos os objectos uni-celulares.

O parâmetro da função *Simul* permite conhecer a posição (acima, abaixo, ou na mesma linha) do objecto que invocou a função. Esta informação é utilizada para evitar ciclos infinitos, já que os objectos podem estar duplamente ligados, e também para fazer actualizações em objectos ligados na vertical.

A seguir apresenta-se um exemplo que ilustra a ordem pela qual são invocadas as funções *Simul* dos objectos existentes num determinado degrau *ladder*.



Neste caso, o CPU principal envia um evento de *Simul* (invocação da função *Simul* do objecto) para o objecto **e**; em seguida, esse evento é enviado para o objecto **c** que, por sua vez, o envia para o objecto **b**, e assim sucessivamente. A função *Simul* do objecto **c** também se encarrega de enviar o evento para o objecto ligado abaixo, **d**. Depois, devido à existência do objecto **d** em *rightObjects*, o CPU principal ainda envia o evento de *Simul* para **d**, só que, tem como resposta um retorno imediato da função *Simul*, pois a função *Simul* do objecto **d** já foi executada no varrimento actual (*flagSimulation == TRUE*).

A função membro *SetDownValSimulation* (*SetUpValSimulation*) serve para actualizar os valores de simulação, *valSimulation*, de todos os objectos situados abaixo (acima) do objecto actual. Esta actualização é necessária pois o valor de simulação em cada nodo é o resultado do OR lógico dos resultados da simulação dos objectos que estão ligados à esquerda desse nodo.

```
BOOL BaseShape :: SetDownValSimulation(BOOL oValue)
{
    if(orDown)
        orDown->SetDownValSimulation(oValue);
    return ( valSimulation = oValue );
}
```

6.4.3. A função virtual de simulação para objectos pluri-celulares

Para objectos com mais do que uma entrada (pluri-celulares), existe a necessidade de redefinir a função virtual *Simul*, de modo a interpretar adequadamente os sinais provenientes dos objectos ligados nas suas entradas.

A classe *CounterReset* representa objectos do tipo Contador Descendente com Reset, pelo que a função *Simul* tem que interpretar dois sinais de entrada: um sinal para decremento do

valor actual do contador (proveniente do objecto ligado à primeira entrada - leftShape[0]) e um sinal de inicialização (proveniente do objecto ligado à segunda entrada - leftShape[1]).

```
BOOL CounterReset :: Simul(WORD flagOnCall)
{
    if (flagSimulation)
        return (valSimulation);        // a simulação já foi feita noutro ramo do circuito
    ...
    if (leftShape[0])                  // verifica se existe objecto ligado à primeira entrada
        counterDown = leftShape[0]->Simul(LEFT_SHAPE_CALL);  // para decrementar o contador
    if (leftShape[1])                  // verifica se existe objecto ligado à segunda entrada
        reset = leftShape[1]->Simul(LEFT_SHAPE_CALL);// reset do contador
    OwnerSimulation();                 // simula a acção do contador
    return (valSimulation);
}
```

A função *Simul* da classe *CounterUpDown* (usada para representar objectos do tipo Contador Ascendente/Descendente com Reset) interpreta mais um sinal do que *CounterReset::Simul*, que é o sinal para incremento do valor actual do contador.

```
BOOL CounterUpDown :: Simul(WORD flagOnCall)
{
    if (flagSimulation)
        return (valSimulation);
    ...
    if (leftShape[0])
        counterUp = leftShape[0]->Simul(LEFT_SHAPE_CALL);      // para incrementar o contador
    if (leftShape[1])
        counterDown = leftShape[1]->Simul(LEFT_SHAPE_CALL);  // para decrementar o contador
    if (leftShape[2])      // reset do contador
        reset = leftShape[2]->Simul(LEFT_SHAPE_CALL);          // para fazer o reset do contador
    OwnerSimulation();
    return (valSimulation);
}
```

Para objectos do tipo Registo de Deslocamento, devem ser interpretados três sinais de entrada: um sinal correspondente ao *bit* a deslocar no registo (*inputVal*); um sinal correspondente ao relógio do registo (*clockOn*); e o sinal de inicialização do registo (*reset*).

```
BOOL ShiftRegister :: Simul(WORD flagOnCall)
{
    if (flagSimulation)
        return (valSimulation);
    ...
    if (leftShape[0])
        inputVal = leftShape[0]->Simul(LEFT_SHAPE_CALL);    // entrada a deslocar no registo
    if (leftShape[1])
        clockOn = leftShape[1]->Simul(LEFT_SHAPE_CALL);    // impulso para deslocar um bit no registo
    if (leftShape[2])
        reset = leftShape[2]->Simul(LEFT_SHAPE_CALL);    // para fazer o reset do registo
    OwnerSimulation();
    ...
    return (valSimulation);
}
```

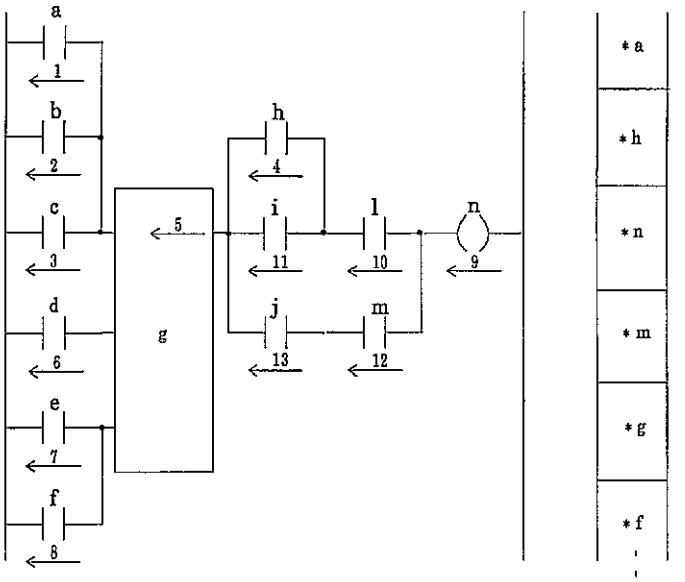
No desenvolvimento do algoritmo de simulação surgiram duas alternativas sobre quais os objectos a guardar no *array rightObjects*:

- guardar somente os objectos terminais (ligados ao fim de continuidade, ou objectos que não tenham nenhuma saída, como o caso dos Registos de Deslocamento);
- guardar todos os objectos posicionados mais à direita em cada linha.

A vantagem da primeira alternativa em relação à segunda é o menor tempo de simulação que lhe está associado. Se analisarmos o exemplo anterior, o CPU principal não necessitaria de enviar o evento de Simul para o objecto **d**. O acréscimo de tempo da segunda alternativa corresponde, no exemplo anterior, ao tempo de uma chamada de uma função virtual, ao teste de uma variável (flagSimulation == TRUE), e ao retorno da função.

A vantagem da segunda alternativa tem a ver com o preenchimento do *array rightObjects*, em que só é necessário detectar os objectos mais à direita, em cada linha do programa *ladder*. Para preencher o *array*, usando a primeira alternativa, era necessário detectar todos os objectos terminais, o que implicaria criar código para distinguir os objectos terminais dos outros objectos. Esse código teria que ser modificado, aquando da introdução de novas instruções, pois haveria que testar se os objectos, representativos das novas instruções, eram ou não objectos terminais. Esta alternativa ia contra aos objectivos de não alteração do código de simulação, no acrescento de novas instruções *ladder* no sistema. Por este facto, decidimos pela segunda alternativa, até porque, o acréscimo de tempo na simulação não nos parece significativo.

À semelhança do exemplo já anteriormente apresentado, a seguir apresenta-se um caso mais elaborado de um degrau que contém um objecto com três entradas, o qual poderá ser, por exemplo, um Contador Ascendente/Descendente. Veja-se a sequência (numerada) do evento de simulação durante um varrimento do degrau *ladder*.



Degrau *ladder* **rightObjects**

6.4.4. A função *OwnerSimulation*

Depois de um objecto "ler" as suas entradas (invocando a função *Simul* para todos os objectos ligados à sua esquerda), este deve invocar a sua própria função *OwnerSimulation* para simular a instrução que representa.

Todas as funções de *OwnerSimulation*, aqui apresentadas, emulam o comportamento das correspondentes instruções, de acordo com o manual de operação do autómato C20H [7].

A função *OpenContact::OwnerSimulation()* implementa a acção de uma instrução *ladder* Contacto Normalmente Aberto. A variável *valSimulation* guarda o estado da simulação do objecto, e *valInSimulation* contém o estado da condição de execução devolvido pelo objecto à sua esquerda. Caso a condição de execução seja TRUE (*valInSimulation = TRUE*), o estado do *bit* de trabalho é carregado para a variável *valSimulation.*

```
BOOL OpenContact :: OwnerSimulation()
{
    flagSimulation = TRUE;          // sinaliza que o objecto actual já foi simulado
    if( valInSimulation )                  // estado da condição de execução da instrução - ON
        return (valSimulation = (*memLocation[0] & maskBit[0]) ? TRUE : FALSE );
    else                                   // estado da condição de execução da instrução - OFF
        return (valSimulation = FALSE );
}
```

A função *CloseContact::OwnerSimulation()* implementa a acção de uma instrução *ladder* Contacto Normalmente Fechado. O comportamento da instrução é em tudo idêntico ao anterior, só que o estado do *bit* de trabalho é complementado antes de ser carregado na variável *valSimulation.*

```
BOOL CloseContact :: OwnerSimulation()
{
    flagSimulation = TRUE;
    if( valInSimulation )
        return (valSimulation = (*memLocation[0] & maskBit[0]) ? FALSE : TRUE );
    else
        return (valSimulation = FALSE );
}
```

As classes *OpenOutput* e *CloseOutput* são as classes representativas de instruções *ladder* que escrevem na memória do PLC virtual, ao contrário das duas anteriores que somente fazem leitura da memória. O código da função *OwnerSimulation*, referente a estas duas classes, é apresentado a seguir:

```
BOOL OpenOutput :: OwnerSimulation()
{
    flagSimulation = TRUE;// sinaliza que o objecto actual já foi simulado
    if(valInSimulation)          // estado da condição de execução da instrução - ON
        *memLocation[0] != maskBit[0];        // coloca o bit de trabalho da instrução no estado ON
    else
        *memLocation[0] &= ~maskBit[0];    // coloca o bit de trabalho da instrução no estado OFF
    return (valSimulation = valInSimulation);
}
```

```
BOOL CloseOutput :: OwnerSimulation()
{
flagSimulation = TRUE;// sinaliza que o objecto actual já foi simulado
    if(valInSimulation)
        *memLocation[0] &= ~maskBit[0];    // coloca o bit de trabalho da instrução no estado OFF
    else
        *memLocation[0] |= maskBit[0];     // coloca o bit de trabalho da instrução no estado ON
    return (valSimulation = valInSimulation);
}
```

Para simular o comportamento de um Temporizador com Atraso à Operação, o objecto *ladder* correspondente tem de conhecer o número de varrimentos (*scan time* - constante ao longo de uma simulação) necessário, para perfazer a temporização, e o número do varrimento actual. A instrução é, então, interpretada da seguinte forma:

- se o valor de entrada (*valInSimulation*) se mantiver no estado ON, só se deve incrementar o número do varrimento (*numberStep*) e compará-lo com o número total de varrimentos (*oTimeInSteps*);
- se todos os varrimentos necessários para atingir o tempo pretendido já foram executados, o estado do *bit* de saída do temporizador é colocado a ON;
- se o estado do sinal de entrada for OFF o temporizador é inicializado (*numberStep = 0*).

```
BOOL TimerOn :: OwnerSimulation()
{
    flagSimulation = TRUE;// sinaliza que o objecto actual já foi simulado
    valSimulation = *memLocation[0] & maskBit[0];// valor, do bit de saída, no varrimento anterior
    if (valInSimulation ) {                          // o estado do valor de entrada (condição de execuçao) é ON
        if( numberStep > oTimeInSteps +1)            // verifica se já se atingiu o fim de temporização
            return valSimulation;                   // já se atingiu o fim de temporização
        if (++numberStep == (oTimeInSteps+1)) {      // incrementa o número de varrimentos e analisa o fim de temp.
            *memLocation[0] |= maskBit[0];           // coloca o bit de saída a ON - indicando fim de temporização
            return (valSimulation = TRUE);           // coloca o valor da simulação do objecto a TRUE
        }
    } else {                                         // o estado do valor de entrada é FALSE
        numberStep = 0;                              // inicializa o temporizador
        *memLocation[0] &= ~maskBit[0];              // coloca o bit de saída no estado OFF
        return (valSimulation = FALSE);              // coloca o valor da simulação do objecto a FALSE
    }
}
```

ConterReset é a classe para representar um Contador Descendente com Reset. Este tipo de contador é utilizado para contar desde um valor inicial até zero. Para cada execução da instrução (*OwnerSimulaton*) o procedimento é o seguinte:

- se o estado da entrada de decremento do contador (*counterDown*) for ON e se esse estado no varrimento anterior (*oldCounterDown*) for OFF, então o valor do contador é decrementado (*--cntCount*);
- se o valor do contador atinge o valor zero, é activado o *bit* de saída do contador;
- se o sinal de *reset* for ON, o contador é imediatamente inicializado e o *bit* da saída é colocado no estado OFF.

```
BOOL CounterReset:: OwnerSimulation()
{
    flagSimulation = TRUE;                  // sinaliza que o contador já foi simulado
    valSimulation = *memLocation[0] & maskBit[0];// valor da simulação é o valor anterior do bit
    if(reset) {                             // a entrada correspondente ao reset do contador foi actuada
        cntCount = Value;                   // coloca o valor inicial no contador
        endOfCount = FALSE;                 // coloca o sinalizador de fim de contagem a FALSE
        *memLocation[0] &= ~maskBit[0];     // coloca o bit de saída a FALSE
        valSimulation = FALSE;              // o valor da simulação é FALSE
        return (valSimulation);
    }
    if( oldCounterDown == counterDown)// o estado da entrada é igual ao estado no varrimento anterior
        return (valSimulation);
    if (counterDown)                    // o estado da entrada é TRUE e, anteriormente, era FALSE
        if(!endOfCount) {               // ainda não se atingiu o fim de contagem
            if(--cntCount == 0) {// decrementa o valor do contador e verifica se se atingiu o fim de contagem
                endOfCount =TRUE;       // sinaliza o fim de contagem
                *memLocation[0] |= maskBit[0];   // coloca o bit de saída a TRUE
                valSimulation = TRUE;
            }
        } else      // fim da contagem
            valSimulation = TRUE;
    oldCounterDown = counterDown;  // guarda o valor para o próximo varrimento do programa
    return valSimulation;
}
```

A acção de objectos do tipo *CounterUpDown* (Contador Ascendente/Descendente com *Reset*) é igual à acção do contador anterior para o sinal de *reset*. Para os sinais de *counterDown* e *counterUp* a acção é a seguinte:

- se os dois sinais são iguais nada acontece;
- quando existe uma transição positiva (passagem do estado OFF para o estado ON), o valor do contador é decrementado ou incrementado, dependendo do sinal de entrada onde houve a transição;
- nos limites, isto é, quando o valor do contador chega a zero ou ao valor inicial do contador, o processamento depende do estado das entradas no varrimento seguinte
    - se o valor for zero e a entrada *counterDown* for actuada, então o valor do contador passa a ser o valor inicial e o bit de saída do contador é actuado;
    - se o valor for o inicial e a entrada *counterUp* for actuada, então o valor do contador passa a ser zero e o *bit* de saída do contador é actuado.

```
BOOL CounterUpDown :: OwnerSimulation()
{
    flagSimulation = TRUE;                          // sinaliza que o contador já foi simulado
    valSimulation = *memLocation[0] & maskBit[0];// valor da simulação é o valor anterior do bit
    if(reset) {                             // a entrada correspondente à entrada de reset do contador foi actuada
        cntCount = 0;                       // inicializa o contador - iguala o valor actual do contador a zero
        endOfCount = FALSE;                 // sinaliza internamente o contador que ainda não terminou a contagem
        *memLocation[0] &= ~maskBit[0];     // coloca o bit de saída a FALSE
        valSimulation = FALSE;              // o valor da simulação é FALSE
        return (valSimulation);
    }
```

```
    if( counterDown && counterUp )  // as duas entradas estão actuadas - pelo fabricante não se deve fazer nada
        return (valSimulation);
    if( oldCounterUp != counterUp ) {      // houve variação da entrada do varrimento anterior para este varrimento
        if (counterUp)                     // a variação corresponde à passagem do estado OFF para o estado ON
            if( endOfCount && cntCount == 0) {  // fim de contagem com o valor actual igual a zero
                ++cntCount;                    // valor incrementado pois houve uma actuação na entrada counterUp
                *memLocation[0] &= ~maskBit[0];   // coloca o bit de saída a FALSE
                valSimulation = FALSE;            // o valor da simulação = FALSE
            }
            else                               // incremento normal do contador - fora dos extremos
                if(++cntCount > Value) {       // incrementa o valor do contador e compara-o com o valor inicial - Value
                    endOfCount = TRUE;         // sinaliza internamente o contador com fim de contagem
                    *memLocation[0] |= maskBit[0];   // coloca o bit de saída a TRUE
                    valSimulation = TRUE;            // o valor da simulação = TRUE
                    cntCount = 0;                    // coloca o valor actual a zero
                }
        oldCounterUp = counterUp;              // guarda o estado actual de counterUp para o varrimento seguinte
    }
    if(oldCounterDown != counterDown) {    // variação, na entrada, do varrimento anterior para o actual
        if (counterDown)                   // a transição da entrada foi positiva
            if( endOfCount && cntCount == Value) {     // fim de contagem com o valor actual igual a Value
                --cntCount;                // valor decrementado pois houve uma actuação na entrada counterDown
                *memLocation[0] &= ~maskBit[0];   // coloca o bit de saída a FALSE
                valSimulation = FALSE;            // o valor da simulação = FALSE
            }
            else
                if(--cntCount < 0) {               // decremento normal do contador - fora dos extremos
                    endOfCount = TRUE;             // sinaliza internamente o contador com fim de contagem
                    *memLocation[0] |= maskBit[0];   // coloca o bit de saída a TRUE
                    valSimulation = TRUE;            // o valor da simulação = TRUE
                    cntCount = Value;                // coloca o valor actual no valor inicial do contador - Value
                }
        oldCounterDown = counterDown;      // guarda o estado actual de counterDown para o varrimento seguinte
    }
    return (valSimulation);
}
```

A função *ShiftRegister:: OwnerSimulation()* simula o comportamento de um Registo de Deslocamento (como o definido em [7]). O objectivo é criar um registo de deslocamento desde a palavra inicial até à palavra final (especificadas nas etiquetas do objecto *ladder*).

- Numa transição positiva do sinal *clockOn*, é feito o deslocamento de um *bit* no registo e é lido o estado do sinal de entrada *inputVal* para o *bit* zero do registo.
- Quando o sinal de *reset* está activo, o registo é inicializado colocando todas as palavras que o definem a zero.

```
BOOL ShiftRegister:: OwnerSimulation()
{
    flagSimulation = TRUE;
    if ( reset) {                  // o sinal de reset está activado
        if ( !oldReset ) {         // o estado anterior do sinal de reset era OFF
            for (WORD i  = 0; i <= numberWords; i++)      // inicialização do registo
                memLocation[0][i] = 0x0000;               // palavra i é inicializada
            oldReset = TRUE;                              // guarda o estado do reset para o varrimento seguinte
        }
```

```
        return FALSE;
    }
    else                        // o estado do sinal de reset é OFF
        oldReset = FALSE;       // guarda o estado do reset para o varrimento seguinte - o estado é FALSE
    if(oldClock != clockOn) {   // houve uma transição no estado do sinal clockOn
        if(clockOn) {           // a transição foi positiva
            // deslocamento da última palavra do registo
            memLocation[0][numberWords] = (memLocation[0][numberWords]) << 1;
            for (INT i = numberWords - 1; i >= 0; i--) {  // deslocamento de todas as outras palavras
                WORD stWord = memLocation[0][i] & 0x8000; // guarda o estado do 16 bit da palavra i do registo
                memLocation[0][i] = (memLocation[0][i]) << 1; // deslocamento de um bit na palavra i do registo
                if ( stWord )                             // o estado do bit guardado é ON
                    memLocation[0][i + 1] |= 0x0001;      // coloca o primeiro bit da palavra i+1 no estado ON
                else
                    memLocation[0][i + 1] &= 0xFFFE;      // coloca o primeiro bit da palavra i+1 no estado OFF
            }
            if (inputVal)                                 // entrada de inputVal no primeiro bit da primeira palavra
                memLocation[0][0] |= 0x0001;              // coloca o primeiro bit da primeira palavra no estado ON
            else
                memLocation[0][0] &= 0xFFFE;              // coloca o primeiro bit da primeira palavra no estado OFF
        }
        oldClock = clockOn;                               // guarda o estado de clockOn para o varrimento seguinte
    }
    return (valSimulation = inputVal);
}
```

6.5. Modos de simulação

Como já foi referido ao longo deste trabalho, foram implementados vários modos de simulação de um programa no PLC virtual. Cada modo de simulação corresponde a um objecto diferente, que é instanciado duma classe pertencente à hierarquia de classes de PLCs virtuais (figura 24). Desta forma, a criação de um novo modo de simulação tem o seu suporte numa nova classe, a definir como uma classe descendente de uma das classes de PLCs virtuais já existentes. As classes terminais da hierarquia, quando instanciadas, dão origem a PLCs virtuais que simulam o programa em modos diferentes.

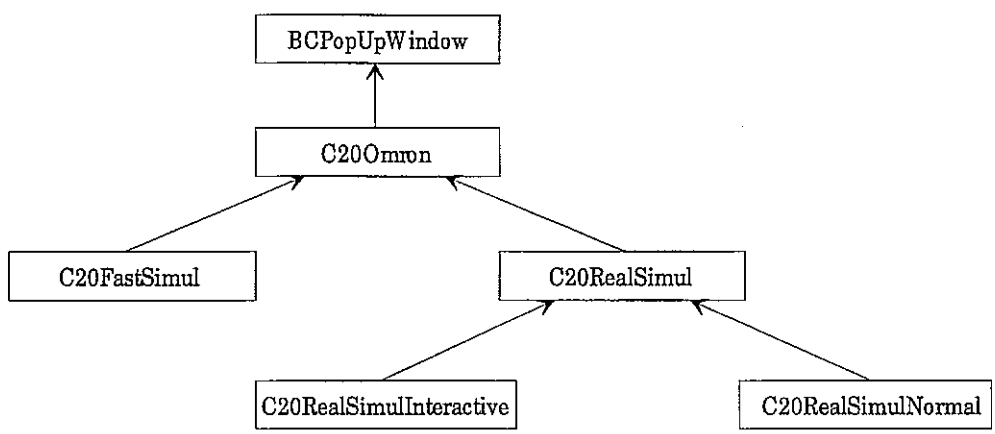


Fig. 24 - Hierarquia de classes representativas dos PLCs virtuais implementados

Os modos de simulação implementados têm as suas características próprias, permitindo diferentes análises para o funcionamento de um mesmo programa. A opção de criar vários modos de simulação (e de dar o suporte à criação de outros modos) foi tomada com o objectivo de dotar o sistema com capacidades acrescidas para o seu uso no ensino de programação e proporcionar diferentes formas de testar programas.

6.5.1. Simulação rápida

O Manual de Utilização, que se encontra em apêndice (ponto A.5.3), apresenta a simulação rápida na óptica do utilizador. Esta simulação define-se como um modo de simulação em que as entradas são introduzidas através do Editor de Estímulos, o programa é simulado sem pausas para todos os pontos definidos no Editor de Estímulos sendo os resultados analisados (após a simulação) usando o mesmo editor. Para construir os estímulos são fornecidas ferramentas (já atrás referidas aquando da apresentação do Editor de Estímulos) que permitem desenhar os estímulos de uma forma rápida e simples. Para analisar os resultados, de modo a verificar a correcção do programa *ladder*, são fornecidas ferramentas de ampliação e de medição de tempos.

Do ponto de vista de programação, este modo de simular o funcionamento do PLC é suportado por um objecto que se instancia da classe *C20FastSimul*. Esta classe é derivada de *C20Omron*, tendo sido redefinidas duas funções virtuais: *InitSimulation* e *UpdateOutputStimulus*.

```
class C20FastSimul : public C20Omron
{
public:
    C20FastSimul(SimulC20Window *opPrnt, SimulationWindow *opSimulWindow);   // construtor da classe
    virtual VOID InitSimulation(VOID);        // para iniciar a simulação
    virtual VOID UpdateOutputStimulus(VOID);      // para actualizar os estímulos
};
```

Em *InitSimulation* é invocada a função *C20FastSimul::InitSimulation()* que inicializa por defeito o objecto, por forma a que este possa simular correctamente o programa. Em seguida, para todos os pontos definidos nos estímulos (*numberOfSteps*), é feita a actualização das entradas, o varrimento do programa e a actualização (e visualização) dos estímulos em simultâneo com a apresentação dos resultados do varrimento actual.

A figura 25 mostra o comportamento da função *InitSimulation* para um exemplo de um programa com uma entrada (*IR_00000*) e uma saída (*IR_00200*). O ponto do diagrama temporal do estímulo *IR_00000*, cujo número de ordem é *stepInSimulation*, é lido pelo simulador (*UpdateInputStimulus*); em seguida é feita uma simulação de um varrimento do programa (*SimulationC20*); por fim, o resultado da simulação é arquivado no ponto com o mesmo número de ordem do diagrama temporal de *IR_00200*. Este processo é repetido desde o número de ordem 0 até *numberOfSteps*.

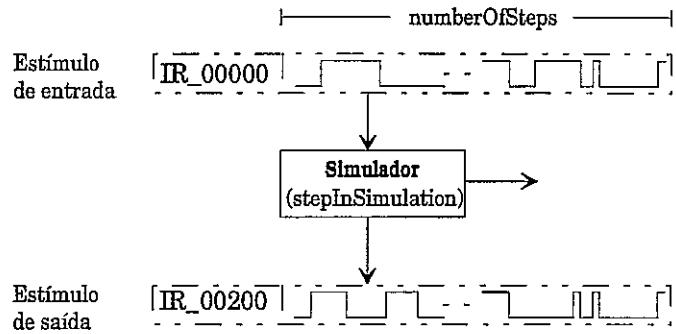


Fig. 25 - Processamento dos pontos dos estímulos pelo simulador

O código para a função *InitSimulation()* é apresentado a seguir. Pode ver-se a codificação da ideia do processamento ilustrado na figura 25. De notar que os varrimentos do programa (*SimulationC20*) são feitos sequencialmente e sem pausas.

```
VOID C20FastSimul :: InitSimulation()
{
    C20Omron::InitSimulation();                    // para a inicialização por defeito
    for (stepInSimulation = 0;stepInSimulation < numberOfSteps; stepInSimulation++) {// para todos os pontos
        UpdateInputStimulus(stepInSimulation);     // inicializa as entradas com os pontos dos estímulos
        SimulationC20();                           // simula um varrimento do programa
        UpdateOutputStimulus(stepInSimulation);// actualiza os estímulos
    }
    ...
}
```

A função *UpdateInputStimulus* é herdada do PLC virtual base, mas a função *UpdateOutputStimulus* é redefinida neste PLC virtual, de modo a permitir a visualização dos resultados na janela do editor de estímulos. Esta janela invoca a função *UpdateOutputStimulus* do PLC base, por forma a guardar os estímulos e, em seguida, desenha o estado do ponto actual para todos os estímulos de saída que estão definidos.

```
VOID C20FastSimul :: UpdateOutputStimulus(VOID)
{
    C20Omron::UpdateOutputStimulus();          // para arquivar os resultados nos estímulos de saída
    int i = 0;
    while(outputStimul[i].outStimul != NULL) {                  // se o estímulo estiver definido
        outputStimul[i].outStimul->DrawBit(stepInSimulation); // desenha os estado do ponto correspondente
        i++;                                                    // incrementa o número de ordem do estímulo
    }
    ...
}
```

6.5.2. Simulação em tempo real

As marcas de tempo na execução de um programa num PLC são o *scan time* (tempo correspondente à execução de um ciclo completo - figura 23), e os tempos envolvidos nos temporizadores eventualmente presentes no programa. Para emular, em tempo real, o

funcionamento de um PLC é necessário garantir que o PLC virtual respeite essas marcas temporais.

A figura 26 ilustra a filosofia do simulador de tempo real desenvolvido, cujo funcionamento pode ser descrito da seguinte forma: o utilizador dá a ordem ao **Simulador** (através de um botão) para que este inicie a simulação em tempo real do programa *ladder*; o **Simulador** procede então às inicializações necessárias e, em seguida, envia um evento de **Início de actividade** para o **Relógio** de simulação, após o que se mantém num estado de espera de eventos de ***Scan Time*** enviados pelo **Relógio;** quando receber um evento o **Simulador** simula um ciclo de execução completo, lendo as **Entradas**, fazendo um varrimento completo e actualizando as **Saídas.**

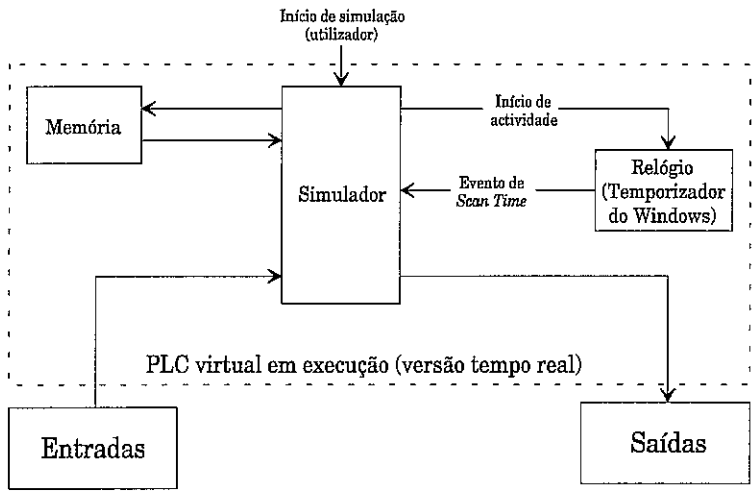


Fig. 26 - PLC virtual em execução (versão tempo real)

Esta simulação é em tudo idêntica à simulação rápida descrita anteriormente, excepto nos instantes em que são iniciadas as simulações dos ciclos, os quais, neste caso, são definidos pelo **Relógio** de simulação.

Pode haver dois factores distintos que impedem que um programa seja simulável em tempo real. Um dos factores tem a ver com o tempo (ts) que o simulador dispende a ler as entradas, a executar um varrimento do programa e a actualizar as saídas. Esse tempo pode ser superior ao periodo do relógio de simulação. Isto leva a que o evento de *Scan Time* não seja imediatamente atendido, sofrendo portanto um atraso que crescerá com o número do ciclo a simular, como se pode detectar na figura 27.

Para verificar se um programa é simulável em tempo real pode utilizar-se uma regra empírica cujo procedimento deve ser o seguinte: simular o programa com o simulador rápido e verificar qual o tempo que o simulador leva para executar todo o processamento necessário. Se esse tempo for inferior a 80% do tempo total de simulação (parâmetro de simulação - Apêndice A, ponto A.4.4.1), então esse programa é simulável em tempo real. Esta regra foi deduzida após vários testes de simulação para vários programas (verificado experimentalmente num PC com a configuração mínima - 386 SX a 16 Mhz).

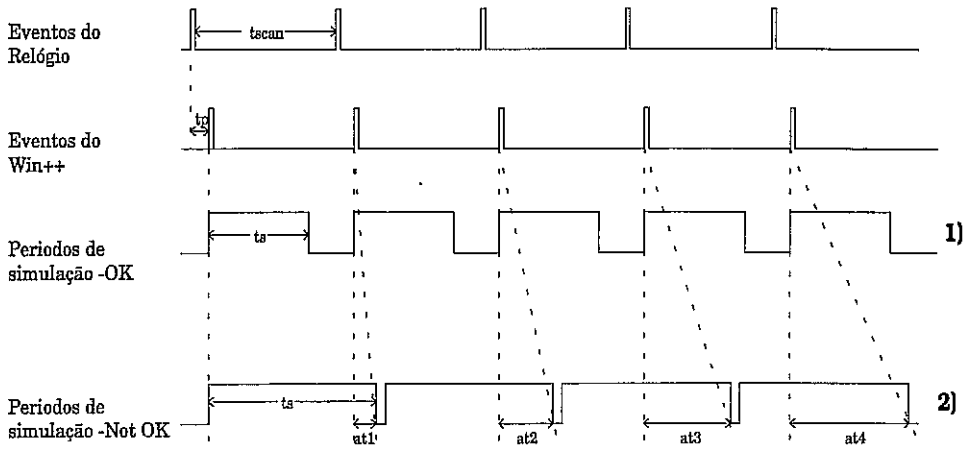


Fig. 27 - Atraso da simulação tempo real provocado pelo tempo de simulação de um ciclo.

tscan - periodo do Relógio de simulação (periodo de tempo entre eventos de *Scan Time*)

tp - tempo de processamento dos eventos de *Scan Time* pelo Windows e Win++

ts - tempo de processamento de um ciclo (*UpdateInputStimulus*, *SimulationC20* e *UpdateOutputStimulus*)

1) - tempo de execução de um ciclo menor que o periodo de Relógio.

2) - tempo de execução de um ciclo maior que o periodo de Relógio.

at1, ..., atn - atrasos na simulação do ciclo **n** relativamente ao evento **n** do relógio de simulação.

Como se pode inferir da figura, quando **tsi** é inferior a **tscan** então a simulação é em tempo real e sem quaisquer atrasos. Porém, quando **tsi** é superior a **tscan** vai existir um atraso dado por

$$\mathbf{at}_n = \sum_{i=0}^{n} (\mathbf{ts}_{i} - tscan), \text{ para } \mathbf{ts}_i \geq \mathbf{tscan}$$

Por exemplo, se Tempo de simulação = 20s; $\mathbf{ts}_i$ = 60ms para qualquer **i**; tscan = 50ms; o tempo de simulação real, em vez de ser de 20s, passará a ser de 24s, havendo portanto um erro de tempo real de 20%. Contudo, convirá referir que os resultados apresentados no Editor de Estímulos não sofrem qualquer tipo de erro, pois a escala temporal é automaticamente adaptada, não dependendo da temporização tempo real.

No caso apresentado anteriormente o erro pode ser quantificável, o que já não acontece na situação descrita a seguir, e que diz respeito aos atrasos introduzidos pelas sobrecargas do Windows. Uma sobrecarga no Windows é traduzida no atraso da resposta a uma determinada mensagem do Windows pela aplicação que consome a mensagem, isto é, ao surgir uma mensagem na fila de mensagens da aplicação, a sua resposta só pode ser dada após algum processamento pendente. Esta situação, no caso das mensagens enviadas por um temporizador, provoca que o seu atendimento pelo PLC virtual não seja imediato (nem tão pouco com um atraso constante, como no caso do exemplo anterior), variando por isso o periodo do relógio da simulação como se pode verificar na figura 28. Como se pode ver, em 2) são executados 4 ciclos de simulação enquanto que em 1), num mesmo intervalo de tempo, são executados 5 ciclos de simulação.

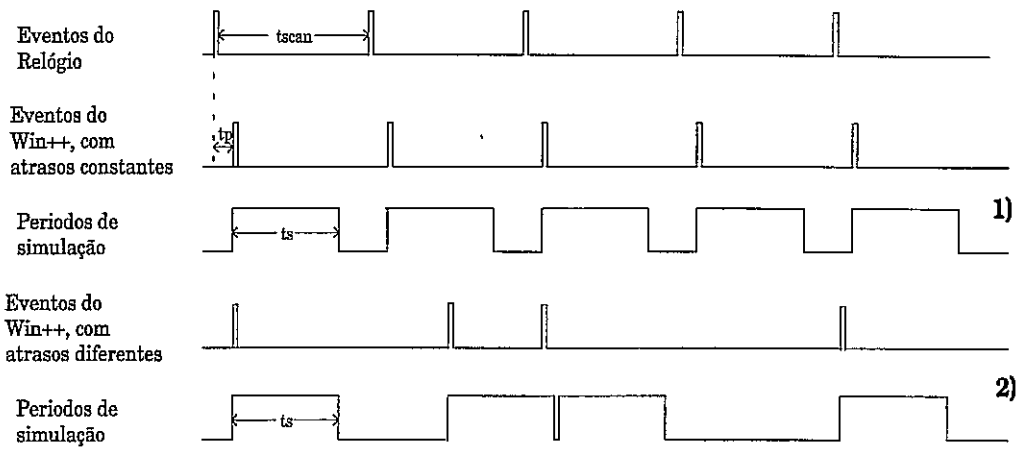


Fig. 28 - Atraso da simulação tempo real, provocado pelo atraso no consumo das mensagens pelo PLC virtual

Esta sobrecarga pode ser provocada por aplicações que estejam a correr em *background* ou pelo atendimento a *interrupts* de *hardware* (gerados, por exemplo, pelo rato ou teclado), já que se garante, através da resposta ao evento *Activaction* (sinaliza a activação ou desactivação de uma janela), que a simulação é imediatamente interrompida quando a janela do PLC virtual fica desactivada.

Em situações de sobrecarga do Windows é impossível quantificar os atrasos **ati** pois estes dependem do número de eventos gerados durante a simulação, que não são necessariamente em igual número para todos os ciclos de simulação.

Uma das possíveis soluções para este tipo de problema consiste em retirar todas as aplicações que estão a correr em *background* no Windows, e evitar gerar eventos de rato e/ou teclado.

6.5.2.1. A classe de representação de temporizadores no Win++

A classe *BCTimerDevice* do Win++ é a interface para as capacidades de temporização do Windows. Objectos instanciados desta classe representam temporizadores, que podem ser programados para gerar um determinado tempo ou para criar um relógio.

No caso presente interessa a segunda hipótese, de modo a criar eventos de *Scan Time*. A mensagem do temporizador é recebida pelo Windows e enviada para o Win++, que a tranforma num evento, através duma chamada à função virtual (*TimerEvent*) da classe janela que contém o temporizador. Depois de criado o temporizador (chamada do construtor de *BCTimerDevice*), podem invocar-se várias das suas funções, nomeadamente a função de início de temporização (*StartTimer*) e a de paragem de temporização (*StopTimer*). O tempo é marcado em milisegundos e é enviado como parâmetro para a função *StartTimer*, contudo a resolução é aproximadamente 55 milisegundos (dados do fabricante do PC), e é afectada pela carga do sistema, como já foi referido anteriormente.

6.5.2.2. A classe base de PLCs virtuais tempo real

A classe base para os PLCs virtuais em tempo real é a classe *C20RealSimulation*, que define o temporizador a utilizar na simulação tempo real. Além disso, esta classe define o código para parar a simulação em tempo real (*StopSimulation*), o código para iniciar a simulação em tempo real e, ainda, o código para parar a simulação quando as condições deixam de ser favoráveis a que esta seja executada em tempo real, isto é, quando a janela do PLC virtual fica inactiva.

```
class C20RealSimul : public C20Omron
{
protected:
    BCTimerDevice *updateStimulusClock;          // relógio de scan time
    virtual VOID Activation(BCEvent *opEvt);     // invocada na alteração da activação da janela
    ...
public:
    C20RealSimul(SimulatorC20Window *opPrnt, SimulationWindow *opSimulWindow);
    virtual VOID StopSimulation(VOID);           // para parar a simulação
    virtual VOID InitSimulation(VOID);           // para iniciar a simulação tempo real
};
```

No construtor da classe, entre outras inicializações, é criado o objecto que serve de relógio da simulação.

```
C20RealSimul :: C20RealSimul(SimulatorC20Window *opPrnt, SimulationWindow *opSimulWindow)
            : C20Omron(opPrnt,opSimulWindow)
{
    ...
    updateStimulusClock =new BCTimerDevice(this, ID_UPDATE_INPUTS_CLOCK, FALSE, GetApp() );
}
```

Na função *InitSimulation* é necessário activar o relógio de simulação, para que este possa gerar eventos de *Scan Time* e enviá-los para a janela do PLC virtual, que depois os processará da maneira adequada.

```
VOID C20RealSimul :: InitSimulation()
{
    StopSimulation(VOID)                                  // para parar a simulação
    C20Omron::InitSimulation();                           // invoca a função de inicialização de defeito
    updateStimulusClock->StartTimer( (WORD)stepPeriod );  // inicializa o relógio de simulação com stepPeriod
}
```

Na função *StopSimulation*, o relógio de simulação é desactivado deixando, por isso, de gerar eventos de *Scan Time*, o que tem como consequência parar a simulação. Além disso, também são inicializados todos os objectos *ladder*, presentes no programa, que dependem do tempo (temporizadores).

```
VOID C20RealSimul :: StopSimulation(VOID)
{
    updateStimulusClock->StopTimer();       // desactiva o relógio de simulação
    Circ->ResetAllTimers();                 // inicializa todos os temporizadores presentes no programa
}
```

A função *Activation* é a função de atendimento a eventos de activação/desactivação da janela. Neste caso, ao detectar-se a desactivação da janela é imediatamente "desligada" a simulação em tempo real pelas razões já anteriormente apontadas.

```
VOID C20RealSimul :: Activation(BCEvent *opEvt)
{
    if(!opEvt->IsActivated() )      // verifica a flag de activação/desactivação proveniente do objecto BCEvent
        StopSimulation();           // se a janela foi desactivada pára a simulação
}
```

A partir da classe base de PLCs virtuais (tempo real) podem derivar-se classes para representar PLCs virtuais para diferentes simulações tempo real. Neste trabalho implementaram-se dois tipos de PLCs virtuais tempo real, que passamos a descrever nos pontos seguintes.

6.5.2.3. A simulação em tempo real normal

A simulação em tempo real normal é caracterizada por os estímulos de entrada serem definidos no editor de estímulos e os resultados da simulação serem apresentados de duas formas: durante o processo de simulação, as entradas e as saídas são apresentadas sobre uma imagem digitalizada do autómato real que está a ser emulado, através de pretensos LEDs à semelhança do que normalmente se encontranos PLCs (ver figura A.15 do Apêndice A, ponto A.5.2.1); após a simulação, todos os resultados podem ser analisados em conjunto, utilizando o Editor de Estímulos.

Para suportar esta simulação derivou-se uma classe (*C20RealSimulNormal*) da classe base de PLC's virtuais tempo real e redefiniram-se algumas funções virtuais. A imagem do PLC real é suportada pelo objecto *c20OmronBitmap*, e as entradas e/ou saídas são representadas por objectos do tipo *BCRectangle*.

```
class C20RealSimulNormal : public C20RealSimul
{
    BCBitmap *c20OmronBitmap;                       // bitmap da imagem do PLC real
    BCRectangle inputRect[MAX_C20_INPUTS];          // rectângulos para representar as entradas do PLC
    BCRectangle outputRect[MAX_C20_OUTPUTS];        // rectângulos para representar as saídas do PLC
    WORD oldInputValue;                             // usado na actualização visual das entradas
    WORD oldOutputValue;                            // usado na actualização visual das saídas
    ...
protected:
    virtual WORD Initialize(VOID);
```

```
    virtual VOID TimerEvent(BCEvent *opEvt);        // atende os eventos enviados pelo relógio de simulação
    BOOL QueryEnd(BCEvent *);                       // invocada no seguimento da acção de Close da janela
public:
    ...
    virtual VOID UpdateInputStimulus(VOID);     // actualiza e visualiza as entradas (provenientes dos estímulos)
    virtual VOID UpdateOutputStimulus(VOID);    // actualiza e visualiza os estímulos e as saídas
    virtual VOID InitSimulation(VOID);          // inicia a simulação
};
```

A redefinição da função *InitSimulation* suporta as acções que se referem especificamente a este objecto, nomeadamente "trazer" a janela do PLC real para o topo e inicializar algumas das variáveis utilizadas no processo de simulação. É, também, invocada a função *InitSimulation* do PLC real base para inicializar o relógio de simulação.

```
VOID C20RealSimulNormal:: InitSimulation()
{
    BringToFront();                 // traz para o topo a janela
    if( !IsVisible())               // verifica se a janela está ou não visível
        Show();                     // mostra a janela caso esta esteja visível
    oldInputValue = 0;
    oldOutputValue = 0;
    ...
    C20RealSimul::InitSimulation();   // inicia a simulação tempo real
}
```

O relógio de simulação, uma vez inicializado, gera eventos de *Scan Time*, o que se traduz na chamada de função virtual *TimerEvent* da classe janela. Como a função *TimerEvent* está redefinida para este objecto, será esta a função invocada. De cada vez que esta função é invocada, executa-se a simulação de um ciclo completo de programa. Além disso, existe a necessidade de testar se a simulação deve ser ou não terminada.

```
VOID C20RealSimulNormal :: TimerEvent(BCEvent *opEvt)
{
    if ( stepInSimulation == numberOfSteps) {    // verifica se já se ultrapassou o número de pontos a simular
        StopSimulation();                        // pára a simulação
        opSimulationWindow->GetSimulationEditor()->Draw();// desenha os resultados no editor de estímulos
        return;
    }
        UpdateInputStimulus(stepInSimulation);      // actualiza as entradas com os estímulos
        SimulationC20();                         // simula um varrimento completo do programa
        UpdateOutputStimulus(stepInSimulation);// actualiza os estímulos de saída e faz a visualização das saídas
        stepInSimulation++;                      // incrementa o número do ponto a simular no próximo varrimento
    ...
}
```

A função *UpdateInputStimulus* inicializa as entradas do PLC tempo real, com os novos estímulos correspondentes ao varrimento actual, e actualiza os "LEDs" correspondentes às entradas em que houve comutação do estado.

```
VOID C20RealSimulNormal:: UpdateInputStimulus(VOID)
{
    C20Omron::UpdateInputStimulus();          // para actualizar as entradas com os novos estímulos
    WORD val = *GetIR(0)^oldInputValue;       // carrega em val as modificações das entradas
    for(int i = 0; i < MAX_C20_INPUTS; i++ )  // para todas as entradas
        if( val & oMaskBit[i] )               // se o estado da entrada i variou
            InvertRect(inputRect[i]);         // inverte o rectângulo correspondente a essa entrada
    oldInputValue = *GetIR(0);              // guarda o estado das entradas para o próximo varrimento
    ...
}
```

A função *UpdateOutputStimulus* actualiza os estímulos de saída, com os resultados do varrimento actual, e actualiza os "LEDs" correspondentes às saídas em que houve comutação do estado.

```
VOID C20RealSimulNormal:: UpdateOutputStimulus(VOID)
{
    C20Omron::UpdateOutputStimulus();         // actualiza os estímulos de saída
    WORD val = *GetIR(2)^oldOutputValue;      // carrega em val as modificações das saídas
    for(int i = 0; i < MAX_C20_OUTPUTS; i++ )
        if( val & oMaskBit[i] )               // se o estado da saída i variou
            InvertRect(outputRect[i]);        // inverte o rectângulo correspondente a essa saída
    oldOutputValue = *GetIR(2);               // guarda o estado das saídas para o próximo varrimento
    ...
}
```

6.5.2.4. A simulação em tempo real interactiva

Define-se a simulação em tempo real interactiva como sendo uma simulação em tempo real em que as entradas são actualizadas pela acção do rato ou teclado, por oposição à simulação tempo real normal em que existe a necessidade de definir os estímulos de entrada no editor de estímulos. De facto, já BARKER [32] preconizava que um modo de simulação interactivo deve permitir ao utilizador estimular directamente o sistema, escolhendo um evento de cada vez.

Importa realçar que este tipo de simulação foi implementado por forma a permitir ao utilizador a construção da sua própria janela interactiva, pelo que esta janela pode variar consoante as necessidades de teste de cada programa. O funcionamento, do ponto de vista do utilizador, pode ser analisado no Apêndice A, ponto A.5.4.3.

O processo que suporta este tipo de simulação é idêntico ao anterior. Corresponde, neste caso, a derivar uma classe (*C20RealSimulInteractive*) da classe base (*C20RealSimul*), a adicionar novos dados e funções membro e a redefinir as necessárias funções virtuais.

```
class C20RealSimulInteractive :  public C20RealSimul
{
    InputOutputRadioButton *opInputs[ MAX_INT_SIMUL_INPUTS];       // botões para as entradas
    InputOutputRadioButton *opOutputs[ MAX_INT_SIMUL_OUTPUTS]; // botões para as saídas/pontos internos
    BCStaticText *opStaticTextInfo;                                // para mostrar informação durante a simulação
protected:
    WORD Initialize(VOID);                  // inicialização dos objectos geridos pela janela interactiva
    VOID TimerEvent(BCEvent *opEvt);        // para atender os eventos do relógio de simulação
```

```
public:
    ...
    virtual BOOL InitSimulation();          // para iniciar a simulação
    virtual VOID StopSimulation(VOID);      // para parar a simulação
    virtual VOID UpdateInputStimulus();     // lê os estímulos para as entradas
    virtual VOID UpdateOutputStimulus();    // actualiza as saídas/pontos internos definidos na janela interactiva
};
```

O objecto instanciado desta classe comporta um conjunto de objectos do tipo *InputOutputRadioButton*. A classe representativa desses objectos é derivada da classe do Win++, fornecida para representar *Radio Buttons*. Assim, estes objectos herdam todas as características desses botões, incluindo as necessárias à gestão de eventos de rato ou teclado; além disso, através de outras características especificamente desenvolvidas, objectos deste tipo permitem fazer a monitorização do estado de *bits* na memória do PLC virtual, bem como fornecer uma plataforma para a modificação do estado das entradas.

```
class InputOutputRadioButton : public BCRadioButton
{
protected:
    char *opAddress;    // endereço do bit que o botão monitoriza ou trabalha (saída/ponto interno ou entrada)
    char *opLabel;      // etiqueta, para representar a entrada ou saída/ponto interno
    WORD *memLocation;  // ponteiro para a palavra que contém o bit na memória do PLC
    WORD maskBit;       // máscara para acesso ao bit
    BOOL isActive;      // sinaliza se o botão está ou não activo
    BOOL isInput;       //sinaliza se o botão representa uma entrada ou uma saída/ponto interno
    BOOL showLabel;     // qual a informação que deve ser visualizada com o objecto (endereço ou etiqueta)
public:
    VOID SetMemLocation(WORD *mem);             // modifica o ponteiro para a palavra em memória
    WORD *GetMemLocation(){return memLocation;} // devolve o ponteiro para a palavra em memória
    BOOL IsInput(){return isInput;}             // devolve a sinalização de entrada ou saída/ponto interno
    ...
};
```

As funções *InitSimulation* e *StopSimulation* apenas invocam as funções idênticas da sua classe antecessora e imprimem mensagens de informação para o utilizador.

```
BOOL C20RealSimulInteractive :: InitSimulation()
{
    opStaticTextInfo->SetText("A simular");     // imprime uma mensagem de sinalização
    C20RealSimul::InitSimulation();             // inicia a simulação tempo real
}
```

```
VOID C20RealSimulInteractive :: StopSimulation(VOID)
{
    C20RealSimul::StopSimulation();             // para parar a simulação
    opStaticTextInfo->SetText("");              // imprime uma mensagem de sinalização
}
```

Na função *TimerEvent* simula-se um ciclo de execução do programa. Neste caso, e ao contrário do PLC tempo real normal, não existe nenhum processamento adicional para parar a simulação, já que esta, não necessitando de definir os estímulos de entrada, não é uma simulação

finita (simulação de um número limitado de pontos). A partir do momento em que a simulação é iniciada, o PLC virtual assemelha-se em tudo a um PLC real em execução, só parando após ordem do utilizador (Opção de paragem de simulação - Apêndice A, ponto A.5.4.2).

```
VOID C20RealSimulInteractive :: TimerEvent(BCEvent *opEvt)
{
    UpdateInputStimulus();      // actualiza as entradas
    SimulationC20();            // simula um varrimento
    UpdateOutputStimulus();     // actualiza os objectos de saída/pontos internos
}
```

A função *UpdateInputStimulus* apenas necessita de ler o estado dos vários botões, representativos dos objectos do tipo entrada, e de actualizar as entradas do PLC virtual.

```
VOID C20RealSimulInteractive :: UpdateInputStimulus()
{
WORD *mem = NULL;
    for(int i=0;i< MAX_INT_SIMUL_INPUTS; i++ ) {  // para todos os objectos que representam entradas
        mem = opInputs[i]->GetMemLocation();      // posição de memória que o objecto opInputs[i] representa
        if (mem)                                  // se o objecto estiver definido
            if ( opInputs[i]->GetState() )        // analisa o estado do botão
    // o estado do botão é ON, então actualiza a entrada correspondente ao objecto opInputs[i] com o estado ON
                *mem |= opInputs[i]->GetMaskBit();
            else
    // o estado do botão é OFF, então actualiza a entrada correspondente ao objecto opInputs[i] com o estado OFF
                *mem &= ~opInputs[i]->GetMaskBit();
    }
}
```

Para cada objecto do tipo saída/ponto interno, a função *UpdateOutputStimulus* lê o estado do *bit* em memória e actualiza o estado do botão correspondente.

```
VOID C20RealSimulInteractive :: UpdateOutputStimulus()
{
WORD *mem = NULL;
BOOL oValue = FALSE;
    // para todos os objectos que representam saídas/pontos internos
    for(int i=0;i< MAX_INT_SIMUL_OUTPUTS; i++ ) {
        mem = opOutputs[i]->GetMemLocation();     // posição de memória que o objecto opOutputs[i] representa
        if( mem ) {                               // se o objecto estiver definido
            oValue=(*mem)&opOutputs[i]->GetMaskBit();  // lê o estado do bit que o objecto opOutputs[i] analisa
            if(!oValue) {                         // se o estado for OFF
                if (opOutputs[i]->GetState() )    // se o estado do botão for ON
                    opOutputs[i]->SetState(FALSE);    // coloca o estado do botão a OFF
            }
            else {                                // o estado do bit é ON
                if (!opOutputs[i]->GetState() )   // se o estado do botão for OFF
                    opOutputs[i]->SetState(TRUE);     // coloca o estado do botão a ON
            }
        }
    }
}
```

Como vimos, o suporte para estas diferentes simulações é bastante simples, como resultado dos mecanismos de herança e de polimorfismo fornecidos pela linguagem C++. De facto, construir um novo modo de simulação corresponde a criar uma nova classe derivada de uma já existente, herdando assim todas as características que permitem simular um PLC, sendo somente necessário definir algumas funções virtuais.

Capítulo 7
Considerações finais

7.1. Modos de simulação do PLC virtual

Uma das vertentes importantes do presente trabalho foi o desenvolvimento de diversos modos para a simulação de programas, os quais, naturalmente, apresentam vantagens e inconvenientes que passaremos a enunciar.

Simulação Rápida

O modo Simulação Rápida permite uma análise bastante detalhada dos resultados de um programa, através da observação de diagramas temporais de entradas e saídas/pontos internos definidos no editor de estímulos. Essa análise é auxiliada por ferramentas fornecidas pelo editor de estímulos, nomeadamente ferramentas de *scroll* e de ampliação, e também de escalas temporais associadas à posição dos cursores.

A possibilidade de criar estímulos apropriados para diversas situações contribui decisivamente para o aumento da capacidade de teste de programas. De facto, podem definir-se estímulos sincronizados ou atrasados no tempo, por forma a simular situações razoavelmente complexas de controlo de processos com múltiplas combinações entre entradas.

Os resultados de programas fortemente dependentes do tempo, em particular os provenientes de processos lentos, podem ser obtidos a uma velocidade muito superior ao tempo real. Assim, devido à rapidez deste modo de simulação, o sistema torna factível a simulação de múltiplas hipóteses para solucionar um mesmo problema.

Como já foi referido anteriormente, é possível armazenar, em disco, conjuntos de estímulos e de características de simulação (como sejam, tempo de simulação, período de varrimento e factor de ampliação) de maneira a possibilitar posteriores simulações ou análises de resultados. Por exemplo, podem escrever-se vários conjuntos de estímulos para um programa (ou para vários programas) e, lançando duas aplicações do PLC virtual, comparar os resultados com base em diferentes janelas do editor de estímulos.

Alguns utilizadores menos experimentados poderão, no início, encontrar algumas dificuldades na interpretação de diagramas temporais. Contudo, com alguma formação e após treino essas dificuldades são rapidamente ultrapassáveis.

A limitação imposta pelo sistema ao tempo de simulação constitui a desvantagem mais evidente do modo Simulação Rápida, pelo que se deve, nos casos em que tal seja problemático, complementar a análise com a Simulação Interactiva, cujo tempo de simulação não é limitado.

Simulação Normal em tempo real

Em aplicações simples, os utilizadores experimentados testam os seus programas através da observação dos indicadores luminosos (correspondentes às entradas e às saídas) existentes no painel do autómato programável. O modo Simulação em tempo real Normal, em virtude de apresentar a imagem do PLC com os respectivos sinalizadores luminosos, permite exactamente o mesmo tipo de observação, ou seja, a verificação em tempo real do funcionamento do programa.

Se os tempos envolvidos não forem demasiado pequenos, este modo de simulação pode ser útil para seguir, em tempo real, o comportamento de temporizadores e contadores existentes no programa, bem como acompanhar a evolução de saídas que deveriam, por exemplo, ser activadas de uma forma sequencial.

Com a Simulação Normal em tempo real não existe a possibilidade de verificar o estado de pontos internos do PLC, à semelhança do que acontece com um autómato real, que só apresenta as entradas e saídas físicas. Contudo, no PLC virtual, os pontos internos podem ser analisados após a simulação recorrendo ao editor de estímulos.

Por ser em tempo real, o uso exclusivo da Simulação Normal no desenvolvimento e teste de soluções, particularmente em processos lentos, pode traduzir-se em tempos de projecto demasiado dilatados.

Simulação Interactiva em tempo real

Como o próprio nome indica, a principal vantagem deste modo de simulação é a interactividade que se consegue utilizando botões que ligam e desligam entradas, à semalhança dos módulos fornecidos por alguns construtores para simular entradas a partir de interruptores mecânicos.

Com a Simulação Interactiva é possível testar equações booleanas, traduzindo-as por degraus de diagramas *ladder*, experimentando várias combinações de entradas e verificando imediatamente o comportamento das saídas.

Este tipo de simulação permite verificar, em tempo real, o estado de pontos internos de um programa, complementando, assim, a Simulação Normal.

Dada a possibilidade de definir etiquetas para os *bits* que se pretendam analisar, a tarefa do utilizador na introdução de dados e na interpretação das saídas fica simplificada. Por outro lado, é permitido ao utilizador definir a posição e o número de objectos na janela interactiva de cada fase de teste de um programa; desse modo, e através de um posicionamento expressivo dos

objectos representativos dos *bits* de entrada ou de outros *bits* a analisar, a interpretação do funcionamento de um programa pode tornar-se mais clara.

À semelhança do que acontece com o editor de estímulos, também aqui é possível gravar em disco as janelas interactivas construídas, de modo a poder executar-se *a posteriori* simulações de um dado programa. Com a criação e gravação de várias janelas interactivas, torna-se fácil avaliar, quase em simultâneo, partes distintas de um mesmo programa.

Finalmente, na Simulação Interactiva, e ao contrário do que acontece na Simulação Rápida, não existe a possibilidade de guardar os resultados da simulação, devido ao facto de se tratar de um modo que não define um tempo finito de simulação.

7.2. Adições ao sistema PLC virtual

O desenvolvimento de todo o sistema PLC virtual foi baseado em metodologias de programação por objectos, as quais permitem a reutilização das classes construídas e simplificam futuras adaptações, nomeadamente a adição de novas instruções *ladder* ao leque das instruções implementadas no PLC virtual, e ainda o desenvolvimento de versões virtuais para outros PLCs reais.

Integração de instruções *ladder*

A integração, no PLC virtual, de novas instruções *ladder* requer um conjunto de acções, quer ao nível da construção das classes que definem essas instruções, quer no que respeita ao desenvolvimento do código que irá permitir a sua inserção na interface gráfica e no módulo de arquivo, que passaremos a apresentar.

A classe representativa de uma nova instrução pode ser derivada de qualquer classe já existente na hierarquia de classes de objectos *ladder*. Por exemplo, se a instrução tiver mais do que uma entrada ou saída, essa classe deve ser derivada da própria classe *Module* (ou de uma sua descendente), para que adquira as características do tipo módulo que a diferenciam de instruções de outro tipo.

Para cada nova instrução, deve ser criado, por exemplo com o *Workshop*, um *bitmap* que lhe confira o aspecto visual que se pretende que aparente. O objecto, do tipo *BCBitmap*, que suporta esse *bitmap* deve ser declarado static, por forma a servir todos os objectos desse tipo eventualmente presentes num programa. Além disso, é necessário redefinir a função virtual *GetBitmap*, que devolve o novo *bitmap*, ou então redefinir a função virtual de *Draw*, se se pretender um *display* diferente para o objecto.

A função virtual *Simul* deve, se necessário, ser redefinida. Tal não é o caso, para novos objectos unicelulares, já que essa função, definida ao nível da classe *BaseShape*, por efeito dos

mecanismos de herança serve todos os objectos desse tipo. Para objectos pluricelulares, pode também não ser necessário redefinir a função *Simul* pois esta, derivada da sua classe antecessora, pode servir os novos objectos, particularmente se eles tiverem o mesmo número de entradas e de saídas que o modelo donde foram derivados.

A função *OwnerSimulation* tem de ser obrigatoriamente redefinida caso a caso, já que é responsável pela acção particular de cada nova instrução. A redefinição de *OwnerSimulation* para uma dada instrução tem de ter em conta os sinais de entrada adquiridos pela função *Simul* para, em seguida e de acordo com as especificações do fabricante do autómato, implementar a instrução.

Para modificar a interface gráfica, de modo a suportar uma nova instrução, só há que acrescentar o código conveniente, não sendo necessário reescrever o que se refere às instruções já existentes.

Para cada nova instrução deve criar-se um *token* (e uma regra) de identificação ao nível do interpretador de reconstrução do programa. Esta adição ocorre nos ficheiros de entrada do *lex* e do *yacc*.

Torna-se também necessário criar uma nova opção na barra de menus do editor de programação, de modo a que o utilizador possa escolher a nova instrução para inserir nos seus programas.

Finalmente, há que acrescentar na função membro *WorkElement*, da classe *ProgramEditor*, o código de criação de um objecto instanciado da nova classe e proceder à sua inserção na grelha. Também, na função *CreatElement*, da classe *MatrixCircuit*, que é invocada pelo construtor do programa *ladder* a partir do arquivo em disco, se deve escrever código para a criar e para inserir objectos deste tipo.

Pode inferir-se da exposição anterior que, para criar uma nova instrução, apenas se torna necessário acrescentar uma classe cujos objectos vão suportar a instrução, e adicionar o código necessário aos mecanismos de gestão da nova instrução na interface. Por outro lado, o código previamente escrito para as outras instruções permanece inalterado e, o que é mais importante, é que a nova instrução pode ser simulada apenas a partir das funções membro da nova classe. Fica, assim, relevada uma das preciosas vantagens da utilização de técnicas de programação por objectos: as facilidades de reutilização de código, originadas pelos mecanismos de herança próprios dessas técnicas.

Emulação de outros PLCs reais

O modelo de virtualização de autómatos programáveis descrito neste trabalho pode ser aplicado a qualquer tipo de PLC, sendo relativamente simples adaptar o *software* desenvolvido de modo a emular outros autómatos comerciais, como se pode verificar pela apresentação seguinte.

A interface gráfica permanece inalterada, mas torna-se necessário modificar a hierarquia de PLCs virtuais e a hierarquia de classes representativas das instruções *ladder*.

As instruções básicas para entradas, saídas e temporizadores são implementadas pelos fabricantes de formas muito similares. Para outras instruções será, eventualmente, necessário desenvolver o código para a função *OwnerSimulation*, por forma a simular o comportamento dessas instruções de acordo com as especificações do fabricante do PLC.

Ao nível da hierarquia dos PLCs virtuais é obrigatório definir a memória para o PLC virtual na classe base de PLCs virtuais, substituir o código da função *GetMemoryAddress*, de modo a suportar o mecanismo de endereçamento do novo PLC, e modificar todas as funções membro que acedem directamente à memória do PLC.

7.3. Conclusões

Neste trabalho foi desenvolvido um sistema que permite virtualizar autómatos programáveis, muito particularmente o modelo C20H da Omron. O sistema foi construído como uma aplicação para o Microsoft Windows e para trabalhar sobre computador pessoal.

Globalmente e do ponto de vista de desenvolvimento, o sistema apresentado pode encarar-se como contendo duas partes distintas: a interface gráfica e o PLC virtual.

Para a implementação da interface gráfica com o utilizador recorreu-se a uma biblioteca de classes em C++ (Win++ da Blaise Computing), a qual permitiu utilizar o Windows numa filosofia de programação por objectos. Os objectos gráficos da interface do PLC virtual foram construídos derivando novas classes, a partir do Win++, às quais foram adicionados dados e funções membro para manipular os novos objectos, e redefinindo algumas funções virtuais de atendimento a eventos.

Outro aspecto importante da interface tem a ver com o uso de *bitmaps* para a representação dos símbolos *ladder*. O utilitário *Workshop* simplificou as tarefas de manipulação de *bitmaps* necessárias à definição da aparência dos símbolos *ladder*, podendo essa aparência ser modificada sem recompilar o programa.

De relevar ainda, ao nível da interface, é a organização dos vários componentes do sistema em janelas autónomas. Esta forma de organização permite ter todo o sistema operacional, nomeadamente o Editor de Programação, o Editor de Estímulos, a janela da Simulação Interactiva e a janela do PLC real (figura 29). Deste modo, o utilizador pode proceder a alterações no programa e, por exemplo, com um simples *click* do rato, simular de imediato o programa num dos três modos de simulação implementados, verificando rapidamente os resultados.

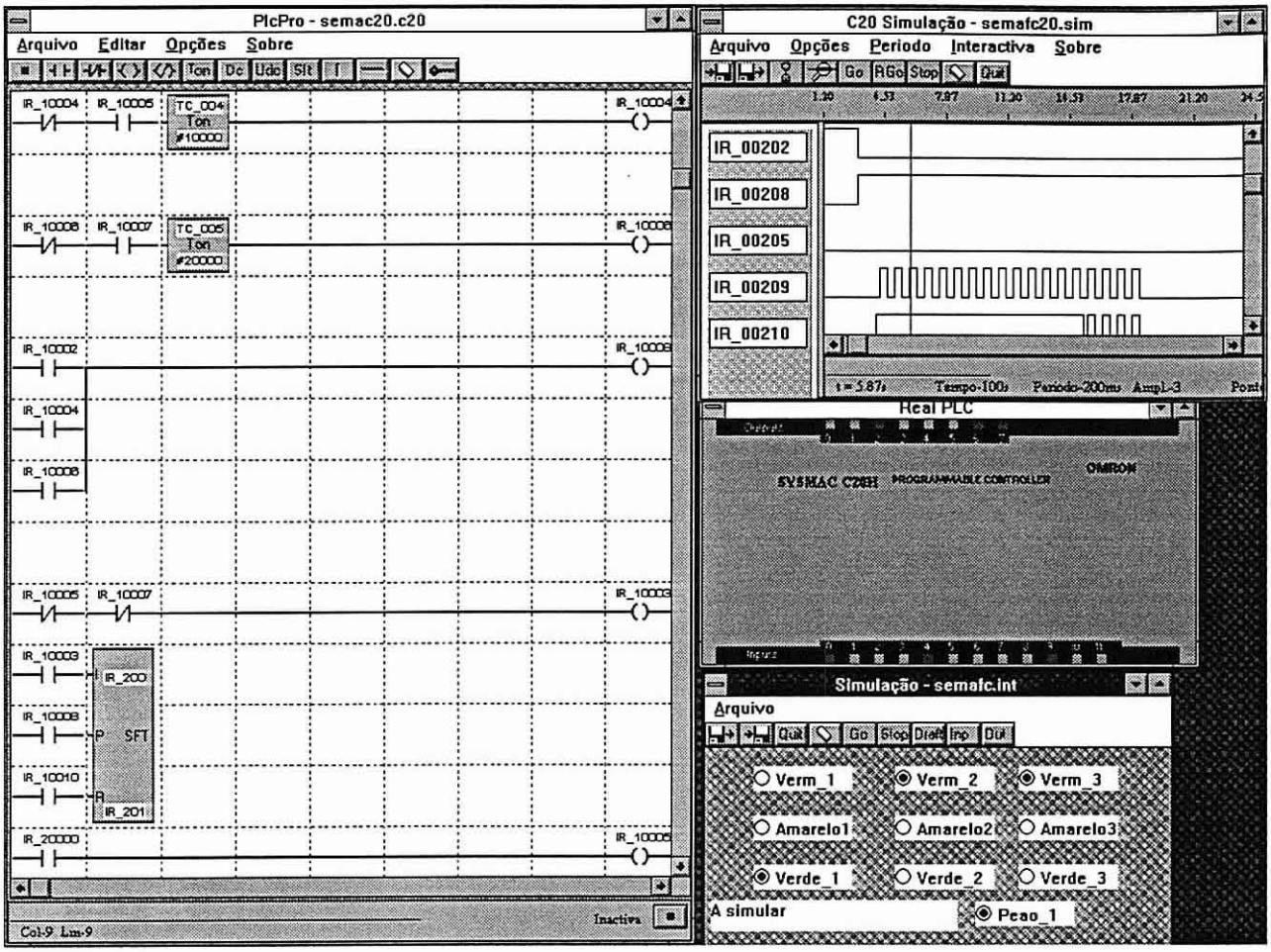


Fig. 29 - Janelas do PLC virtual

A construção do PLC virtual tem por base um conjunto de classes, desenvolvidas em C++, que permitem representar quer os PLCs virtuais, quer as instruções que eles reconhecem. Assim, foram criadas duas hierarquias: uma para suportar todos os PLCs virtuais e a outra para suportar as instruções *ladder* que o PLC virtual reconhece. Para cada hierarquia foi definida uma classe abstracta de base, de modo a definir as funcionalidades e características a que todas as classes descendentes devem obedecer. Desta forma, a criação de novas instruções e/ou de novos PLCs virtuais encontra-se "regulamentada" pelas classes ascendentes, e corresponde à derivação de uma classe a partir de uma das hierarquias.

Para a simulação de programas desenvolveu-se um algoritmo genérico, implementado com base nos mecanismos de herança e de polimorfismo das linguagens de programação por objectos. O algoritmo de simulação traduz a forma como é interpretada a linguagem *Ladder*, dotando os vários objectos *ladder*, presentes num dado programa, com competências de simulação, ou seja, cada objecto *ladder* interpreta as suas ligações a outros objectos e simula a instrução que representa na memória do PLC virtual.

O sistema visa dois tipos de utilização: como auxiliar de ensino para cursos iniciais sobre programação de PLCs; como ferramenta para programação e diagnóstico de aplicações

envolvendo PLCs. O sistema foi já utilizado em módulos didácticos de dois cursos de formação profissional, tendo permitido cumprir cabalmente a leccionação desses módulos, sem recorrer a laboratórios equipados com autómatos reais e respectivo software específico e/ou consolas de programação. Por outro lado, uma versão preliminar do sistema PLC virtual é usada no centro de desenvolvimento, na Alemanha, de um fabricante de PLCs, para treino dos seus programadores, pois verificaram que o modo de simulação rápida é muito elucidativo e eficaz na detecção de certos erros na fase de desenvolvimento de programas.

Como foi referido anteriormente, para a construção do sistema foi desenvolvido um modelo virtual para um autómato programável, o qual, muito embora tenha sido concretizado para um PLC em particular, pode ser adaptado para outros autómatos, já que a maioria dos cerca de 70 ficheiros desenvolvidos, contendo mais de 11000 linhas de código, são reutilizáveis. Além disso, o PLC virtual é expansível, nomeadamente ao nível dos módulos de entrada e saída de dados, já que o modelo prevê a possibilidade da adição de outros blocos.

7.4. Trabalho futuro

Um trabalho da índole do que foi apresentado é necessariamente uma peça incompleta e imperfeita, que requer alterações que a melhorem e complementem.

O sistema PLC virtual deveria ser testado com um maior número de utilizadores, de modo a inferir sobre possíveis modificações principalmente ao nível da interface gráfica.

Outras possibilidades para prosseguir este trabalho, por forma a aumentar as suas capacidades, são apresentadas a seguir:

- Desenvolver uma placa para comunicação do PLC virtual com o exterior, o que corresponde a construir um sistema de entradas/saídas semelhante ao existente nos PLCs reais. Tal placa permitiria a inserção do sistema PLC virtual em situações reais de controlo.

- Incorporar no PLC virtual outros editores de programação, que permitam programar o PLC virtual noutras linguagens, principalmente nas definidas no recente *standard* (IEC 1131 Standard - Part 3 Programming Languages, 1993), o qual engloba duas linguagens gráficas (Linguagem *Ladder* e Diagramas de Blocos Funcionais), duas linguagens textuais (Lista de Instruções e Texto Estruturado) e uma linguagem gráfica com uma estrutura baseada no *Grafcet* (*Sequential Function Chart*).

- Estabelecer mecanismos de conversão entre as diversas linguagens.

Referências bibliográficas

[1] PICARD, Robert P., Gordon J. Savage - **Instructing Industrial Controls Using Ladder Diagrams on an IBM PC.** In: IEEE Transactions on Education, vol. E-29, no. 1, February 1986.

[2] FALCIONE, Albert, Bruce H. Krogh - **Design Recovery for Ladder Logic.** In: IEEE Control Systems, April 93.

[3] KNON, Wook Hyun, Jong-il Kim, Jaehyun Park - **Architecture of a Ladder Solving Processor (LSP) for Programmable Controllers.** In: Proceedings of IECON'91 - International Conference of Industrial Electronics, Control and Instrumentation. Kobe (Japan), 1991.

[4] HUGHES, Thomas A. - **Programmable Controllers.** Instrument Society of America (ISA), 1989.

[5] MICHEL, Gilles - **Programmable Logic Controllers.** John Wiley & Sons, 1990.

[6] LYTLE, Vincent A. - **A Survey of PLC Language Features.** In: Proceedings of the Seventh Annual Control Engineering Conference. Rosemont (USA), June 1988.

[7] OMRON Corporation - **Mini H-type Programmable Controllers - Operation Manual.** Omron, July 1990.

[8] WHITE, Robert B., R. K. Read, M. W. Koch, R. J. Schilling - **A Graphics Simulator for a Robotic Arm.** In: IEEE Transactions on Education, vol. 32, no. 4, November 1989.

[9] NORTON, Peter, Paul Yao - **Borland C++ Programming for Windows.** Bantam Books, 1992.

[10] Microsoft Corporation - **Windows Version 3.0 - Software Development Kit - Manuals.** Microsoft Corporation, USA 1990.

[11] STROUSTRUP, Bjarne - **What is Object-Oriented Programming ?.** In: IEEE Software, May 1988.

[12] MEYER, Bertrand - **Reusability: The Case for Object - Oriented Design.** In: IEEE Software, March 1987.

[13] THOMAS, Dave - **What's an Object ?.** In: Byte, March 1993.

[14] MEYER, Bertrand - **Object oriented software construction.** Prentice-Hall International (UK), 1988.

[15] STROUSTRUP, Bjarne - **The C++ programming language.** Addison-Wesley, 1986.

[16] STROUSTRUP, Bjarne - **A better C ?.** In: Byte, August 1988.

[17] CUTRONA, Louis J. - **Class Conflit**. In: Byte, September 1991.

[18] BLAISE Computing - **WIN++, class library for Windows**, Version 2.1. Blaise Computing, Inc., 1991.

[19] Borland - **Borland C++ 3.1, Manuals.** Borland International, USA 1991.

[20] AZEVEDO, J. L., J. A. Ferreira, J. P. Estima de Oliveira - **Universal System to Program and Simulate Programmable Controllers**. In: Actas das 3as Jornadas Hispano-Lusas de Ingeniería Eléctrica. Barcelona (Espanha), July 1993.

[21] LESK, E., E. Schmidt - **Lex: A Lexical Analyser Generator**. In: The Unix Programmer's Manual, Suplementary Documents. 1975.

[22] JONHSON, Stephen C. - **YACC: Yet Another Compiler-Compiler**. In: The Unix Programmer's Manual, Suplementary Documents. 1975.

[23] BIEN, Christopher - **Simulation a necessity in safety engineering.** In: Robotics World, December 1992.

[24] RAZ, Tzvi - **Graphics Robot Simulator for Teaching Introdutory Robotics**. In: IEEE Transactions on Education, vol. 32, no 2, May 1989.

[25] WILLIAMS, Tom - **Object oriented methods transform real time programming**. In: Computer Design, September 1992.

[26] SWAINSTON, Fred - **A system approach to programmable controllers**. Delmar Publishers Inc., 1992.

[27] BABB, P. - **Is Ladder Logic The Right Software Tool ?**. In: Control Engineering, June 1988.

[28] MYERS, Brad A. - **Taxonomies of visual programming and program visualization**. In: Journal of Visual Languages and Computing, March 1990.

[29] BISCHAK, Diane P., Stephen D. Roberts - **Object Oriented Simulation**. In: Proceedings of the 1991 Winter Simulation Conference. Phoenix (USA), December 1991.

[30] FERREIRA, J. A., J. L. Azevedo, J. P. Estima de Oliveira - **Virtualization of Programmable Logic Controllers**. In: Proceedings of the International AMSE Conference on Systems Analysis, Control & Design - SYS' 93. London (England), September 1993.

[31] ESTIMA DE OLIVEIRA, J. P., J. L. Azevedo, J. A. Ferreira, P. J. Ferreira - **Software Development for Programmable Logic Controllers - a methodology and a system**. In: Proceedings of the IFAC Workshop on CIM in Process and Manufacturing Industries. Espoo (Finlândia), November 1992.

[32] BARKER, A., J. Song - **A Graphical simulation Tool for programmable logic controllers**. In: IEE Colloquium on Discret Event Dynamic Systems - Generation of Modelling, Simulation and Control Applications. London, 1992.

Apêndice A

Manual de Utilização da aplicação PLC virtual (PlcPro)

Abril 94

Índice - Apêndice A

Capítulo A.1
A.1 Introdução

Desde o seu aparecimento na década de 60, os autómatos programáveis têm vindo a dar um contributo fundamental para o aumento de produtividade nas indústrias, principalmente nas indústrias de produção.

Hoje em dia cada vez mais empresas, nas mais variadas áreas, usam este tipo de computadores industriais para automatizar as suas linhas de produção, para controlar máquinas e processos, etc.. Por este facto, e devido à complexidade crescente deste tipo de aplicações, aumentou a necessidade de formar mais técnicos especializados na programação de autómatos.
O sistema aqui descrito (denominado PlcPro) permite a instalação com baixo custo de laboratórios para o ensino da programação de autómatos industriais, também designáveis abreviadamente por PLCs (do inglês "Programmable Logic Controllers").

A.1.1 Características do sistema

A ideia subjacente ao sistema PlcPro é a construção de um PLC por *software*, de forma a que as entradas são obtidas com o rato ou com o teclado, e as saídas são visualizadas no monitor. Este sistema é constituído por uma aplicação que corre em ambiente Windows e que representa a virtualização de um autómato programável.

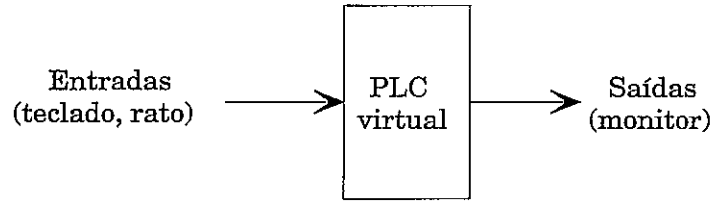


Fig. A.1 - Comunicação com o PLC virtual

Esta virtualização é completa e permite a escrita e a execução de programas desenvolvidos em Linguagem de Contactos. Depois de escrito o programa, o funcionamento do autómato pode ser simulado em três modos diferentes: em tempo real, simulação rápida e simulação interactiva. Cada uma destas simulações tem as suas vantagens nas diversas formas de teste dos programas.

Os vários autómatos existentes no mercado são conceptualmente muito similares, quer no que diz respeito ao modo de funcionamento, quer na maneira de os programar. Contudo há diferenças fundamentais entre as marcas, as famílias e mesmo os modelos de autómatos disponíveis; essas diferenças traduzem-se, por exemplo, no número de entradas e saídas, número

máximo de passos de programação e, principalmente, nos endereços físicos atribuídos pelo fabricante aos diferentes componentes que constituem um PLC.

Assim, o presente sistema foi desenvolvido tendo como objectivo a simulação do modelo C20H da OMRON.

A.1.2 Hardware e software necessários

A configuração mínima de *hardware* é constituída por computador pessoal PC AT 386Sx, equipado com disco rígido, 2 MB de RAM e rato.

Para correr a aplicação PlcPro são necessários os ficheiros fornecidos e uma versão do Microsoft Windows (3.0 ou 3.1) instalada no computador.

A.1.3 Interface gráfica

A interface gráfica de todo o sistema tem por base a interface fornecida pelo Microsoft Windows. A escolha desta interface justifica-se pelas suas características amigáveis e pela imensa divulgação no mundo dos computadores pessoais; o seu uso permite a novos utilizadores do sistema uma rápida aprendizagem da interface com o utilizador. Minimizar o tempo de aprendizagem da interface é essencial de modo a aproveitar o máximo tempo disponível para o que realmente é importante que, neste caso, é aprender a programar autómatos.

O sistema oferece um editor de programação em Linguagem de Contactos, um editor de estímulos de simulação e um editor próprio para a simulação interactiva.

A.1.4 Organização deste manual

Este manual está organizado em seis capítulos. No capítulo A.1 faz-se uma breve introdução de modo a informar qual o âmbito de utilização desta aplicação. No capítulo A.2 encontra-se o necessário para a instalação completa da aplicação PlcPro.

Nos outros capítulos procede-se a uma análise mais detalhada de todos os componentes da aplicação. Assim, são apresentados nos capítulos A.3 e A.4 todas as funcionalidades dos editores de programação e de estímulos, respectivamente. No capítulo A.5 são apresentados os vários modos oferecidos pelo PlcPro para a simulação de programas. No capítulo A.6 são detalhadas as sequências de operações necessárias à criação de um programa com o editor de programação; também neste capítulo se explicita o *modus operandi* essencial à simulação de programas previamente editados.

Capítulo A.2
A.2 Instalação

Este capítulo explica como proceder para a correcta instalação do sistema PlcPro.

Nota importante: para a instalação é necessário dispôr de, pelo menos, 1.5 MB livres no disco rígido. Naturalmente que, para desenvolver novos programas, mais espaço em disco deve ser libertado.

A.2.1 Tipos de ficheiros

Os ficheiros necessários são apresentados a seguir:

wpp_b311.dll	livraria
wpp_b312.dll	livraria
bwcc.dll	livraria
plcpro.exe	programa executável
*.c20	ficheiros de programas - exemplos
*.sim	ficheiros de estímulos - exemplos
*.int	exemplos de janelas interactivas

A.2.2 Instalação no disco rígido

Juntamente com o sistema, é fornecido o programa (install.exe) que permite a instalação automática de todos os ficheiros necessários. A instalação mais correcta corresponde à colocação de todos esses ficheiros no mesmo directório. Deve então proceder da seguinte forma:

1- Criar o directório C20, digitando as seguintes duas linhas:

md c:\c20

cd c:\c20

2- Correr o programa *install.exe*. Inserir a disquete e escrever:

a:install

Após a instalação e a escolha da resolução do monitor, o ficheiro de demonstração, correspondente à resolução não seleccionada, pode ser apagado.

A.2.3 Criação do ícone de programa

Para a criação do ícone de programa (supondo a versão Inglesa do Windows) devem seguir-se os passos:

1- Criar uma janela de grupo

- Seleccionar o menu *FILE* do *Program Manager*
- Seleccionar o sub-menu *NEW*
- Seleccionar a opção *Program Group*
- Na opção *Description* escrever por exemplo PlcPro, e premir no botão *OK*

2- Seleccionar novamente *FILE* e em seguida *NEW*

3- Seleccionar a opção *Program Item*

4- Preencher a caixa de diálogo apresentada na figura A.2 e, em seguida, seleccionar OK.

Deverá aparecer o ícone de programa como se mostra na figura A.3.

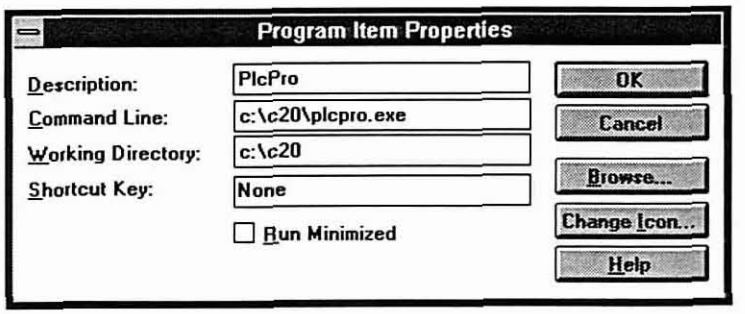


Fig. A.2 - Caixa de diálogo para criação do ícone da aplicação

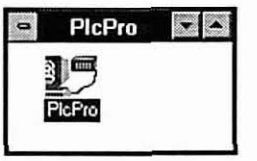


Fig. A.3 - Ícone do PLC virtual

Está agora em condições de iniciar a aplicação PlcPro.

Capítulo A.3
A.3 Editor de programação

O editor de programação permite a escrita de programas em Linguagem de Contactos. Esses programas poderão posteriormente ser simulados pelo PLC virtual.

A.3.1 Janela do editor

O editor encontra-se na janela principal da aplicação e possui uma grelha constituída por células, na qual vão ser colocados símbolos gráficos. A interligação adequada desses símbolos (na horizontal e na vertical), permite representar um programa em Linguagem de Contactos. A figura A.4 mostra a janela principal e a grelha. Nesta versão do PlcPro a grelha é composta por 9x50 células.

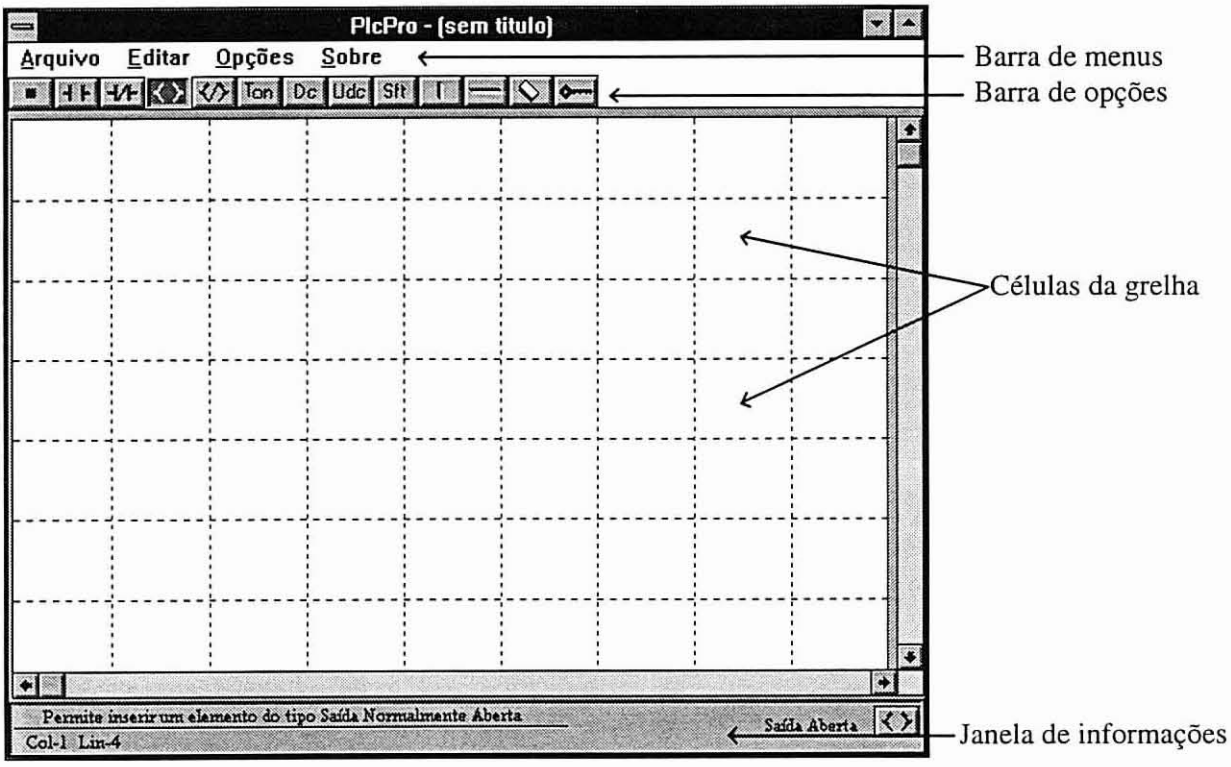


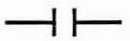
Fig. A.4 - Editor de programação

O editor fornece ferramentas para inserir e apagar símbolos na grelha, e permite utilizar os comandos tradicionais de Copiar, Cortar, Apagar e Colocar. Além disso, permite também a edição dos símbolos, de modo a modificar os parâmetros que os definem ou caracterizam.

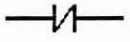
A.3.2 Símbolos gráficos da Linguagem de Contactos

Estão implementados símbolos para representar contactos, saídas, contadores, temporizadores e registos de deslocamento.

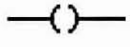
Os símbolos implementados correspondem às instruções básicas (em diagramas de contactos) utilizadas praticamente por todos os construtores de autómatos. De agora em diante, e por uma questão de facilidade de apresentação, os símbolos gráficos serão divididos em dois grupos: símbolo tipo elemento - os primeiros quatro da tabela 1; e símbolos tipo módulo - os restantes.



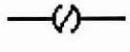
Contacto normalmente aberto. O endereço do *bit* (condição lógica) a testar é o único parâmetro necessário para símbolos deste tipo.



Contacto normalmente fechado. O endereço é idêntico ao do contacto normalmente aberto.

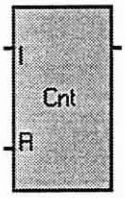


Saída normalmente fechada. O endereço representa também o endereço da saída ou ponto interno.

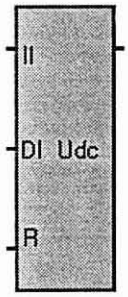


Saída normalmente aberta. O endereço representa também o endereço da saída ou ponto interno.

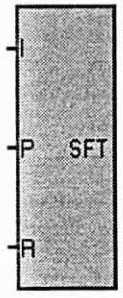
Temporizador com atraso à operação. São dois os parâmetros para este símbolo: o endereço do *bit* de saída do temporizador, isto é, o *bit* que sinaliza o fim da temporização; e o valor que representa o tempo pré-programado em milésimos de segundo.



Contador descendente. Este símbolo tem duas entradas tipo *bit* sendo uma a entrada do impulso de contagem (I) e a outra uma entrada de inicialização do contador (R). Existem dois parâmetros a definir para este símbolo: o endereço do *bit* de saída do contador, isto é, o *bit* que sinaliza o fim de contagem; e a constante que representa o valor inicial do contador.



Contador ascendente/descendente. Este símbolo tem três entradas tipo *bit*: a entrada de impulso de contagem ascendente (II), a entrada do impulso de contagem descendente (DI) e uma entrada para inicialização do contador (R). Existem dois parâmetros a definir para este símbolo: o endereço do *bit* de saída do contador, isto é, o *bit* que sinaliza o fim de contagem; e a constante que representa o valor inicial do contador.



Registo de deslocamento. Este símbolo tem três entradas do tipo *bit*: a entrada a deslocar pelo registo (I), uma entrada para impulso de deslocamento (P) e uma entrada de inicialização do registo (R). Para este símbolo existem dois parâmetros tipo endereço, que indicam o endereço inicial e o endereço final das palavras de 16 *bits* que vão constituir o registo.

Tabela 1

A.3.3 Menus e opções

Os menus e as opções permitem dar a funcionalidade desejada à aplicação. A distinção que é feita tem a ver com a aparência de ambos, isto é, os menus apresentam-se de uma forma textual, enquanto que as opções se apresentam de uma forma iconizada. Neste ponto faz-se uma descrição de cada um dos menus e opções:

A barra de menus mostra os menus existentes no editor de programação. Cada menu tem um conjunto de sub-menus:

Arquivo

Novo	Cria um novo programa
Abrir	Carrega um programa para o editor
Gravar	Grava o programa do editor para o disco
Gravar como	Grava o programa com um outro nome
Sair	Abandona a aplicação
Sobre PlcPro	Informação

Editar

Cortar	Apaga os símbolos seleccionados e copia-os para o *clipboard*
Copiar	Copia os símbolos seleccionados para o *clipboard*
Colocar	Coloca os símbolos que estão no *clipboard* na posição seleccionada
Apagar	Apaga os símbolos seleccionados

Opções

Grelha	Permite mostrar/esconder a grelha de suporte do editor de programação
Redesenhar	Permite desenhar o programa presente no editor
Inserir Linha	Permite a inserção de uma nova linha no programa (pelo menos uma célula pertencente à linha, a partir da qual se vai inserir a nova linha, deve estar seleccionada)
Bloquear Opção	Bloqueia o mecanismo de opções. Se a opção não estiver bloqueada, na sequência de uma qualquer acção na grelha o editor coloca-se num estado inactivo (vide opção Inactiva). Se o mecanismo estiver bloqueado, qualquer opção permanece seleccionada até que nova opção seja escolhida. Na janela de informação existe a informação de bloqueio de opção
Editor de Texto	Mostra e torna activa a janela do editor de texto
Simulação	Mostra e torna activa a janela do editor de estímulos

Sobre

Info	Informação

São as seguintes as opções fornecidas pelo editor de programação:

Inactiva	A opção Inactiva permite ao utilizador editar e seleccionar símbolos já inseridos na grelha (a selecção de símbolos é necessária para a utilização dos comandos do menu Editar)
Contacto Aberto	Permite inserir um contacto normalmente aberto
Contacto Fechado	Permite inserir um contacto normalmente fechado
Saída Aberta	Permite inserir uma saída normalmente aberta
Saída Fechada	Permite inserir uma saída normalmente fechada
Temporizador	Permite inserir um temporizador de atraso à operação
Contador Desc.	Permite inserir um contador descendente
Contador Asc./Desc.	Permite inserir um contador ascendente/descendente
Registo Desl.	Permite inserir um registo de deslocamento
Ligação Vertical	Permite inserir uma ligação vertical
Ligação Horiz.	Permite inserir uma ligação horizontal
Borracha	Permite apagar qualquer símbolo
Bloqueio	Tem o mesmo significado que o sub-menu Bloquear Opção.

A.3.4 Janela de informações

A janela de informações apresenta informações importantes para o utilizador, nomeadamente: sobre o tipo de interacção que o rato vai ter no editor, através da indicação da opção seleccionada, e sobre o bloqueio do mecanismo de opções. Além disso, indica qual a coluna e a linha da célula de trabalho activada, e fornece uma ajuda automática na escolha das opções e/ou menus.

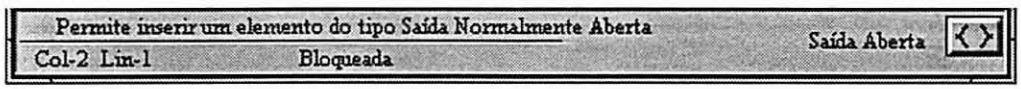


Fig A.5 - Janela de informações do editor de programação

A.3.5 Janela do editor de texto

A finalidade deste editor de texto é a visualização dos arquivos dos programas escritos no editor de programação, já que o arquivo é uma tradução fiel da grelha do editor. A forma de arquivo dos programas em modo texto permite a escrita dos programas com um simples editor de texto, possibilitando ao utilizador a escrita de programas noutro computador mesmo que não possua o Windows. Este editor é também útil para copiar linhas de outros programas para o

programa presente no editor de programação, sem haver a necessidade de carregar o dito programa para o editor de programação; para tal basta abrir o ficheiro de arquivo com o editor de texto, seleccionar da forma tradicional as linhas a copiar e, em seguida, usar o sub-menu "Colocar" na posição desejada do editor de programação.

Os comandos deste editor são idênticos aos usados normalmente nos editores de texto.

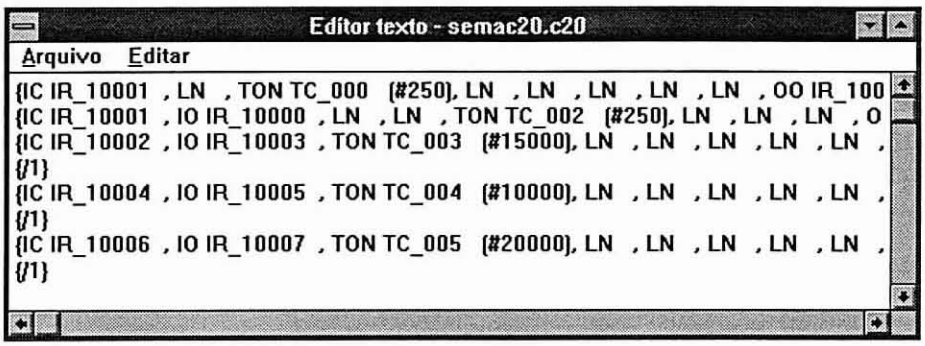


Fig. A.6 - Janela do editor de texto com o arquivo de um programa *semac20*

A.3.6 Escrita de programas em Linguagem de Contactos

Neste ponto serão dadas todas as informações necessárias para o desenvolvimento de programas em Linguagem de Contactos usando o editor de programação, mas pressupondo que o utilizador está familiarizado com a estrutura dessa linguagem. Contudo, algumas regras de construção deste tipo de linguagem são verificadas pelo editor, como por exemplo:

- na última coluna do editor somente poderão existir saídas, não permitindo o editor inserir outros símbolos;

- as saídas só poderão ser inseridas na última coluna e não em qualquer outra.

A.3.6.1 Inserir símbolos

Para inserir um determinado símbolo na grelha, deve proceder-se da seguinte forma:

- Seleccionar a opção que corresponde ao símbolo desejado.
- Premir o rato sobre a célula onde se pretende colocar o símbolo.

Quando a opção seleccionada permite inserir um símbolo, o cursor do rato é representado por um lápis, enquanto que para a inserção de uma ligação vertical o cursor do rato é representado por uma linha vertical. Para a opção Inactiva o cursor é representado por Edit .

A.3.6.2 Apagar símbolos

Para apagar símbolos pode-se proceder de duas formas diferentes, dependendo do número de símbolos a apagar: utilizando o processo associado ao menu Editar/Apagar, ou utilizando a opção (Borracha).

O menu Apagar é descrito no ponto A.3.6.4; no segundo caso o procedimento deve ser o seguinte:

• Seleccionar a opção  da barra de opções (o rato é agora representado por uma borracha).

• Premir o botão do rato sobre a célula onde se encontra o símbolo a apagar.

A.3.6.3 Editar parâmetros de símbolos

Um símbolo, além de ter uma representação gráfica na Linguagem de Contactos, contém vários parâmetros que o identificam, dependendo o número de parâmetros do tipo de símbolo em questão. Por esse motivo, existem dois tipos de caixas de diálogo para edição dos referidos parâmetros, podendo estes ser endereços ou valores numéricos. Os endereços representam posições de memória, sendo possível endereçar um *bit* ou uma palavra de 16 *bits*. Um valor numérico pode representar, por exemplo, o tempo pré-programado de um temporizador ou o valor inicial de um contador.

Para editar os parâmetros de um símbolo o utilizador deve proceder da seguinte forma:

•Seleccionar a opção  (Inactiva).

•Premir duas vezes o botão do rato sobre o símbolo a editar.

Dependendo do tipo do símbolo aparece uma das duas caixas de diálogo seguintes:

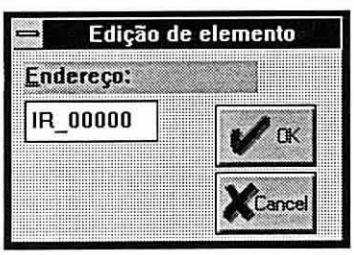


Fig. A.7 - Edição de um elemento

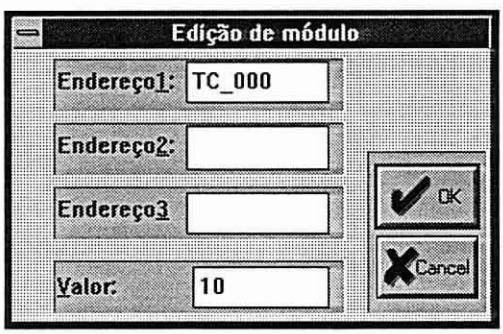


Fig. A.8 - Edição de um módulo

A caixa da figura A.7 permite editar um só endereço, enquanto que a caixa da figura A.8 permite editar no máximo três endereços e um valor. A edição destas etiquetas é feita normalmente, ou seja, com os comandos existentes para as caixas de diálogo.

A.3.6.4 Copiar, cortar e apagar símbolos

Para copiar símbolos dentro de um programa deve-se:

1- Seleccionar o(s) símbolo(s) a copiar

• Premir o botão do rato sobre a célula a copiar(ou usar o rato numa acção tradicional de arrastamento (*drag*) sobre as células a copiar).

- Escolher o sub-menu Copiar do menu Editar.

2- Seleccionar a célula destino do primeiro símbolo a copiar

- Premir o botão do rato sobre a célula pretendida.

3- Escolher o sub-menu Colocar do menu Editar.

Para usar o comando Cortar o processo é semelhante ao anterior, mas substituindo a escolha do sub-menu Copiar pelo sub-menu Cortar.

Para usar o comando Apagar basta seleccionar os símbolos a apagar e escolher o sub-menu Apagar.

A.3.6.5 Gravação e leitura de programas

Os comandos implementados para a leitura (gravação) de programas de (no) disco são os tradicionais. Para qualquer deles é criada uma caixa de diálogo de modo a facilitar as acções do utilizador. Os menus são:

Novo Cria um novo programa

Abrir Carrega um programa para o editor de programação

Gravar Grava o programa presente no editor para o disco

Gravar como Permite gravar o programa presente no editor com outro nome

Como já se referiu no ponto A.3.5, os ficheiros de arquivo são em formato de texto.

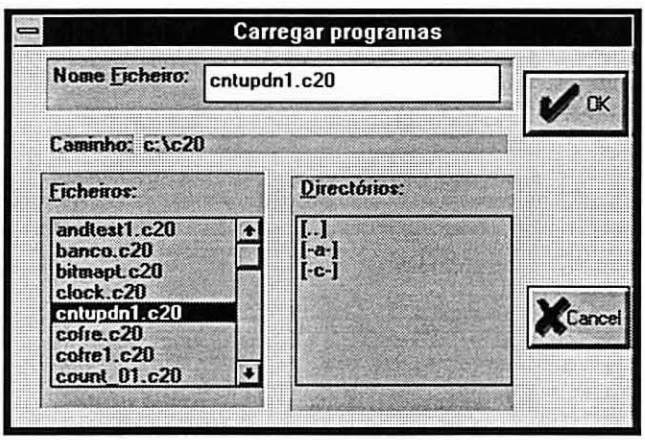


Fig. A.9 - Caixa de diálogo para carregar programas

Capítulo 4
A.4 Editor de estímulos

Como é sabido, um autómato industrial, em função da leitura do estado das suas entradas e de acordo com o programa de controlo residente em memória, comanda as suas saídas. Existe então a necessidade de introduzir as entradas para o PLC virtual e de verificar o comportamento das saídas, por forma a avaliar o bom ou o mau funcionamento do programa. Com base nessa avaliação, o programador procederá, se for caso disso, às necessárias correcções.

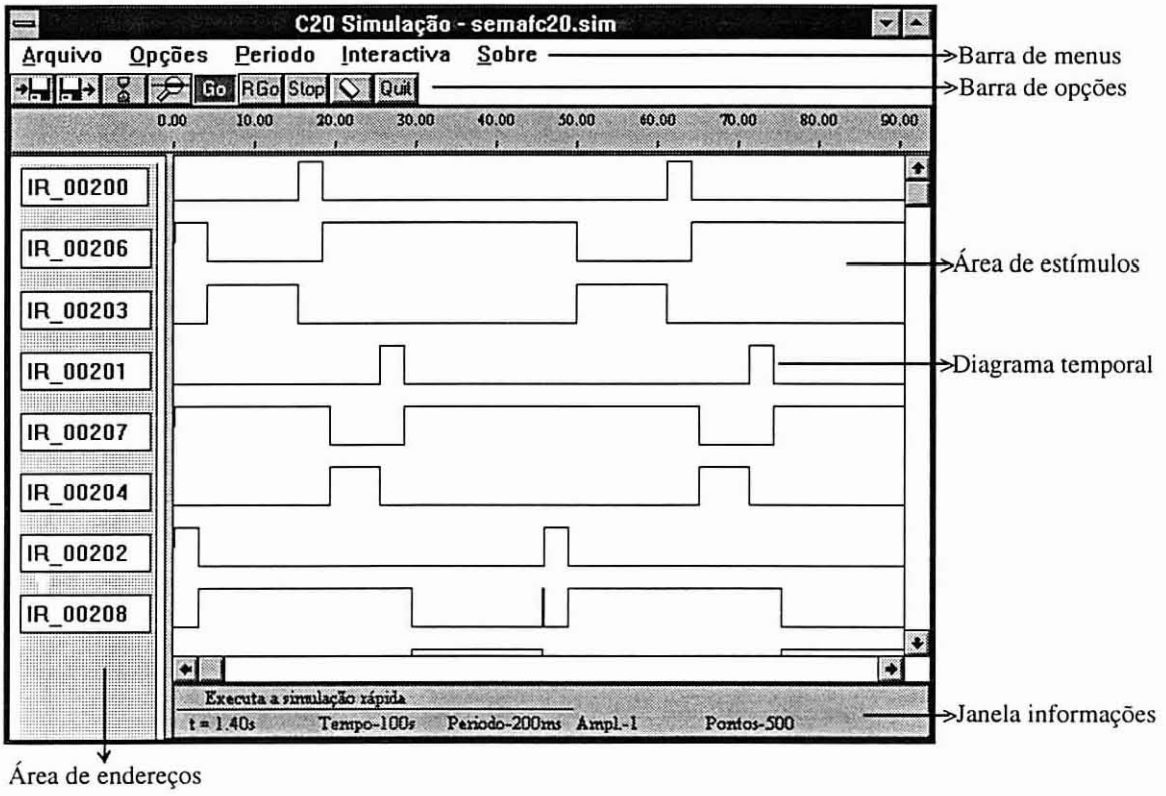


Fig. A.10 - Editor de estímulos

Para fazer face ao problema apresentado, oferece-se um editor para a criação de estímulos que traduzam os estados das entradas do PLC virtual. Este editor constitui também, como se verá mais tarde, um meio de visualização dos resultados de alguns dos modos de simulação implementados, pois permite mostrar os estados das saídas e dos pontos internos.

O editor de estímulos encontra-se numa janela própria que pode ser activada por meio do sub-menu Simulação, contido no menu Opções do editor de programação.

A.4.1 Opções e menus

Na barra de menus, o menu Arquivo contém os sub-menus tradicionais, pelo que apenas os outros são aqui apresentados.

Opções

Redesenhar Permite redesenhar todo o editor de estímulos.

Linha no rato Transforma o cursor do rato numa linha vertical para analisar com maior detalhe as transições no diagrama temporal.

Período

Este menu permite escolher o período para a simulação em tempo real. Aqui, o período determina a taxa a que cada ciclo do programa é executado. No início de cada execução é lido um conjunto de estímulos de entrada, e no fim de cada execução são gerados os sinais resultantes.

50 ms O período de varrimento de todo o programa é feito com intervalos de 50 milisegundos.

100 ms Idem, 100 milisegundos.

...

500 ms Idem, 500 milisegundos.

Interactiva

Int_Simul Mostra a janela de simulação interactiva.

Na barra de opções encontram-se as seguintes facilidades:

 **Abrir** Lê estímulos, pré-gravados no disco, para o editor.

 **Gravar** Grava o conteúdo do editor de estímulos para disco.

 **Tempo** Permite definir o tempo de simulação (expresso em segundos) até um máximo de 300 segundos. Para inserir o tempo existe uma pequena caixa de diálogo:

Fig. A.11 - Inserção do tempo de simulação

Ampliação Permite definir o factor de ampliação do editor de estímulos.

S. Rápida Executa a simulação rápida, se não houver erros no programa. Havendo erros no programa, eles são sinalizados através de mensagens.

S. Real Executa a simulação em tempo real.

Stop Interrompe a simulação em tempo real. A simulação em tempo real também é interrompida automaticamente se o utilizador provocar alguma acção que não permita o correcto funcionamento desta simulação como, por exemplo, escolher menus ou mudar de janela.

Borracha Permite apagar completamente um estímulo, incluindo o diagrama temporal e o endereço.

Sair Esconde a janela do editor de estímulos.

A.4.2 Definição de endereços

Os endereços são escritos, pelo utilizador, na área respectiva (área de endereços), e servem para: identificar as entradas que é necessário definir antes da simulação; identificar as saídas ou os pontos internos que se pretendem visualizar após a simulação. Cada endereço identifica um só *bit,* ao qual pode corresponder uma entrada, uma saída ou um ponto interno.

Para cada entrada deve definir-se o endereço e o diagrama temporal correspondente.

No fim da simulação, o editor mostra os diagramas temporais das saídas ou dos pontos internos correspondentes aos endereços definidos na área de endereços.

Para posicionar o cursor para escrita dos endereços deve usar-se o rato, enquanto que o deslocamento para outros endereços pode ser feito usando o rato ou o teclado (através das teclas *TAB* ou *SHIFT TAB*)

A.4.3 Definição dos estímulos das entradas

A definição do diagrama temporal dos estímulos das entradas é feita utilizando o rato sobre a área de estímulos do editor. Na figura A.12 mostra-se um estímulo, que é composto por um endereço e pelo seu diagrama temporal.

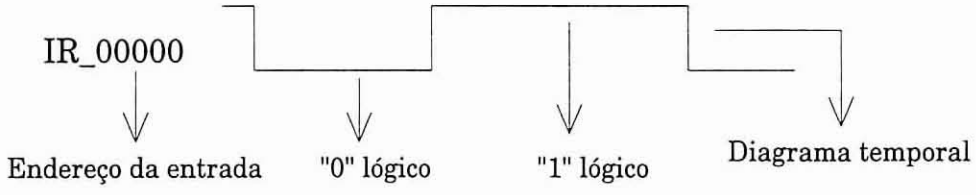


Fig. A.12 - Composição de um estímulo

O diagrama temporal pode ser traçado ponto a ponto (premindo o botão do rato nas posições adequadas) ou usar o método de arrastamento do rato se se pretende uma definição mais rápida. Para definir um "1" lógico, o cursor do rato deve estar acima da linha média, enquanto que para obter um "0" o cursor deve estar abaixo da linha média.

A ampliação (vide A.4.4.2) e a escala podem facilitar o desenho dos diagramas temporais.

Para apagar completamente um estímulo, deve-se seleccionar a opção  e em seguida premir o rato sobre o diagrama temporal do estímulo a apagar.

A.4.4 Facilidades adicionais

Para ajudar o utilizador e tornar o editor mais flexível são fornecidas outras facilidades.

A.4.4.1 Definição de parâmetros

Podem-se definir os seguintes parâmetros:

- O tempo de simulação (Tempo) - define o tempo total da simulação. Pode ser modificado pelo utilizador utilizando a opção  .
- O período do relógio de simulação (Período) - define o instante de cada execução do programa. Pode-se escolher um dos períodos do menu Período.

Na janela de informações mostra-se outro parâmetro que é o número máximo de pontos do diagrama temporal (Pontos); é calculado, tendo em conta os dois parâmetros anteriores, através da fórmula Pontos = Tempo / Período.

A.4.4.2 Ampliação

A ampliação pode ser alterada de duas formas. Uma delas faz uso da opção  , que cria uma caixa de diálogo onde se pode inserir o factor de ampliação. O factor de ampliação estabelece o número de pixeis usados na representação de um ponto, até um máximo de 60 pixeis.

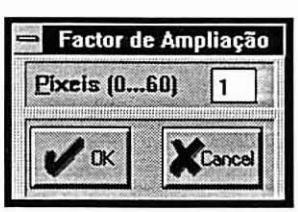


Fig. A.13 - Inserção do factor de ampliação

A outra forma de modificar o factor de ampliação, obtém-se seleccionando o menu Linha no rato e utilizando o rato para escolher a área a ampliar: premir o rato no início da área e, em seguida, premir novamente o rato no fim da área pretendida. Desta forma consegue-se visualizar a área escolhida em toda a dimensão da janela, modificando assim o factor de ampliação.

A.4.4.3 Janela de informações

Esta tem a mesma função da janela de informações do editor de programação.

Fig. A.14 - Janela de informações do editor de estímulos

A informação apresentada no exemplo acima tem o seguinte significado:

t = 21.80s A posição actual do rato corresponde a um tempo de 21.80 segundos
Tempo-100s O tempo de simulação é de 100 segundos
Período-200ms O período pré-programado é 200 milisegundos
Ampl.-1 O factor de ampliação é 1 pixel por ponto
Pontos-500O número total de pontos a simular é 500

A.4.4.4 Gravação e leitura de estímulos

Os comandos disponíveis para a gravação e leitura de programas são os tradicionais:

Novo Cria um novo conjunto de estímulos
Abrir Carrega um conjunto de estímulos para o editor de estímulos
Gravar Grava um conjunto de estímulos para o disco
Gravar como Permite gravar o conjunto de estímulos com outro nome

Capítulo 5
A.5 Simulação

O programa presente no editor de programação pode ser simulado em vários modos, possibilitando um teste alargado ao seu funcionamento. Esses métodos de simulação são aqui designados por "simulação em tempo real", "simulação rápida" e "simulação interactiva". Este capítulo descreve o conjunto de operações a efectuar para a introdução, simulação e visualização de dados em cada um dos modos referidos.

A.5.1 Parâmetros de simulação

Os parâmetros a definir para a simulação são dois e já foram referidos no ponto A.4.4.1: o tempo de simulação (Tempo) e o período do relógio da simulação (Período).

A.5.2 Simulação em tempo real

Como o próprio nome indica, esta simulação permite a execução em tempo real do programa, respeitando desta forma os tempos associados aos temporizadores e os instantes exactos em que as entradas mudam de estado.

A.5.2.1 Janela do autómato real

Esta janela aparenta a imagem do autómato real, onde as entradas e as saídas são mostradas usando os tradicionais sinalizadores, mais conhecidos por *LED's*.

É possível desta forma verificar as mudanças de estado das saídas e das entradas, à medida que o programa vai sendo executado.

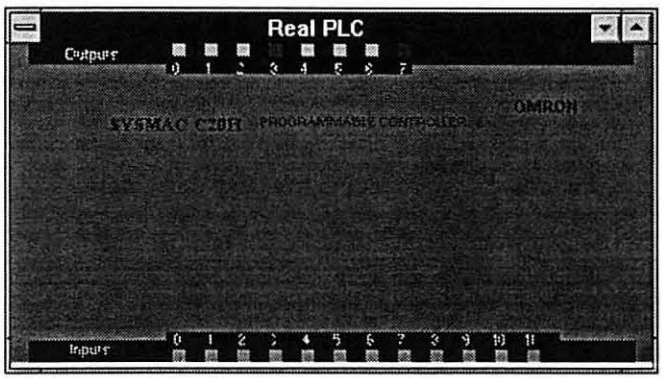


Fig. A.15 - Janela do autómato real

A.5.2.2 Visualização dos resultados

Os resultados podem ser visualizados durante ou no fim da simulação. Durante a simulação a visualização dos resultados é feita sobre a janela do autómato real. Após a simulação, a janela do autómato real é escondida automaticamente, sendo os resultados visualizáveis sobre a janela do editor de estímulos que, entretanto, se torna activa (veja-se a figura A.10).

Para utilizar o editor de estímulos é necessário definir os endereços de *bit* das saídas e/ou dos pontos internos que se querem observar no final da simulação. Esses endereços são editados na área de endereços.

No final da simulação, os diagramas temporais dos estímulos das entradas, e os resultantes do processo de simulação, podem ser analisados em detalhe utilizando as facilidades de ampliação e de *scroll.*

A.5.3 Simulação rápida

A simulação rápida permite executar o programa sem pausas, isto é, após um varrimento do programa (o qual corresponde a um ciclo de execução do autómato real) passa-se de imediato para o varrimento seguinte. Este modo de simulação é de extrema utilidade quando se está a testar um programa fortemente dependente do tempo e se pretende observar os resultados rapidamente, continuando a garantir uma excelente precisão na medida.

A visualização dos resultados é idêntica à da simulação em tempo real, quando se usa o editor de estímulos. Assim remete-se o leitor para o capítulo A.4 onde esse editor é descrito, e para o parágrafo A.6.5.1 onde se apresenta um exemplo ilustrativo.

A.5.4 Simulação interactiva

Esta simulação não necessita da definição prévia dos estímulos de entrada através do editor de estímulos. Durante a simulação, os estímulos das entradas são introduzidas pelo utilizador, usando o rato sobre uma janela construída para o efeito. Os resultados são também visualizados nessa janela.

A.5.4.1 Janela de simulação interactiva

Com o menu Interactiva do editor de estímulos é possível criar uma janela interactiva. Esta janela permite a inserção de objectos que representam entradas, saídas ou pontos internos, de acordo com as necessidades de teste. Cada objecto contém um endereço e uma etiqueta, podendo o utilizador escolher a etiqueta ou o endereço para ser visualizado com o objecto. Por exemplo, na figura seguinte, para todas as saídas, só são visualizadas as etiquetas.

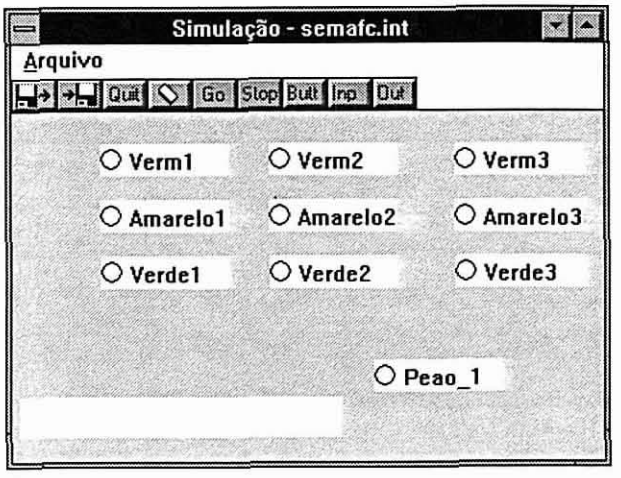


Fig. A.16 - Janela interactiva

A.5.4.2 Opções e menus

O único menu disponível, neste modo de simulação, é o Arquivo, o qual é idêntico aos menus Arquivo dos editores de programação e de estímulos.

As opções são as apresentadas a seguir:

 **Abrir** Lê uma janela interactiva previamente gravada em disco.

 **Gravar** Grava a janela interactiva para o disco.

 **Sair** Esconde a janela do editor de estímulos.

 **Borracha** Permite apagar objectos, que representam entradas, saídas ou pontos internos.

 **Simular** Permite executar a simulação interactiva.

 **Stop** Interrompe a simulação interactiva. Esta, à semelhança da simulação em tempo real, também é interrompida automaticamente se o utilizador executar alguma acção que não permita o correcto funcionamento da simulação como, por exemplo, escolher menus ou mudar de janela.

 **Botões**

 **Rascunho** Só uma destas duas opções está activa em cada momento. Quando a opção  está visível é permitido configurar a janela interactiva. Quando a opção  está visível, é possível executar a simulação interactiva. Como estas opções ocupam a mesma posição na barra de opções, em cada momento só uma estará visível.

 **Entrada** Insere um objecto do tipo entrada.

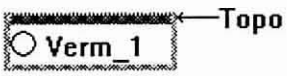
 **Saída** Insere um objecto do tipo saída (ou ponto interno).

A.5.4.3 Configuração da janela interactiva

Como se pode ver na figura 16 oferece-se a possibilidade de colocar os objectos (entradas ou saídas) em qualquer posição na janela; esta pode também ter as dimensões e a localização que o utilizador desejar. Isto permite que, para cada programa, o utilizador coloque as entradas e/ou as saídas nas posições mais adequadas aos seus propósitos de teste. Por exemplo, para um pequeno programa de controlo de semáforos de trânsito, as saídas poder-se-iam organizar como se mostra na figura A.16.

Para deslocar um objecto de uma posição para outra deve:

- Seleccionar a opção Butt / Draft de modo a que Butt fique visível.
- "Pegar" o objecto no topo e deslocá-lo para a nova posição, isto é, premir o rato sobre o topo e arrastar o rato com o botão premido até ao local desejado, deixando só então de premir o botão.



A.5.4.3.1 Criação das entradas

A janela interactiva permite no máximo 12 entradas distintas. Para criar uma entrada deve:

- Seleccionar a opção Butt
- Seleccionar a opção Inp

Como resultado, aparece um objecto que representa uma entrada e situado numa determinada posição. O objecto pode, então, ser deslocado como se explicou no ponto anterior.

A.5.4.3.2 Criação das saídas ou pontos internos

A janela interactiva comporta no máximo 18 saídas (ou pontos internos) distintas. Para a criação de uma saída/ponto interno deve:

- Seleccionar a opção Butt
- Seleccionar a opção Out

Como resultado, aparece um objecto que representa uma saída/ponto interno e situado na posição programada por defeito. Naturalmente que, também é possível reposicionar o objecto.

A.5.4.3.3 Edição dos endereços e/ou etiquetas

Cada um dos objectos, sejam eles do tipo entrada ou do tipo saída, necessitam pelo menos de um endereço para representarem uma entrada ou saída/ponto interno do PLC virtual. Para a edição do endereço ou da etiqueta deve:

- Seleccionar a opção Butt
- Premir o rato duas vezes seguidas sobre o objecto a editar.

Aparece, então, a caixa de diálogo (figura A.17) na qual vão ser escritos o endereço e a etiqueta desejados. Além disso pode-se escolher, através dos botões Etiqueta ou Endereço, qual dos dois deve ser visualizado em conjunto com o objecto. No caso da figura A.17 é a etiqueta que vai ser visualizada juntamente com o objecto.

A caixa de diálogo informa sobre o tipo de objecto editado (Tipo), o qual, no exemplo apresentado abaixo, é do tipo saída/ponto interno.

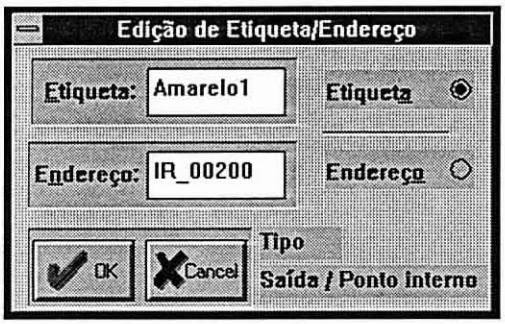


Fig. A.17 - Edição de objectos da janela interactiva

A.5.4.4 Interacção com a simulação

Para iniciar o processo de simulação interactiva deve seleccionar a opção Go.

A interacção do utilizador faz-se utilizando o rato. Esta interacção permite modificar o estado lógico das entradas; para tal, basta premir o botão do rato sobre o objecto que representa a entrada desejada.

Os resultados da simulação são ilustrados através da mudança de estado das saídas (ou pontos internos) previamente definidas sobre a janela interactiva.

A.5.4.5 Gravação e leitura do aspecto da janela

Os comandos implementados para a leitura e gravação de programas são os tradicionais:

Novo Cria uma nova janela interactiva
Abrir Lê uma janela interactiva do disco
Gravar Grava uma janela interactiva para o disco
Gravar como Permite gravar uma janela interactiva com um outro nome

Capítulo A.6
A.6 Criação e simulação de um programa

Neste capítulo, e partindo de um pequeno exemplo, descrevem-se as acções necessárias para escrever e simular programas utilizando o sistema PlcPro.

Exemplo - Construir um relógio com uma entrada de Início. O relógio deve ter um período de 1 segundo e um tempo de serviço de 50%.

A.6.1 Escrita do programa

O programa apresentado na figura seguinte é uma das soluções possíveis para o problema proposto. Comecemos então, por escrevê-lo em Linguagem de Contactos:

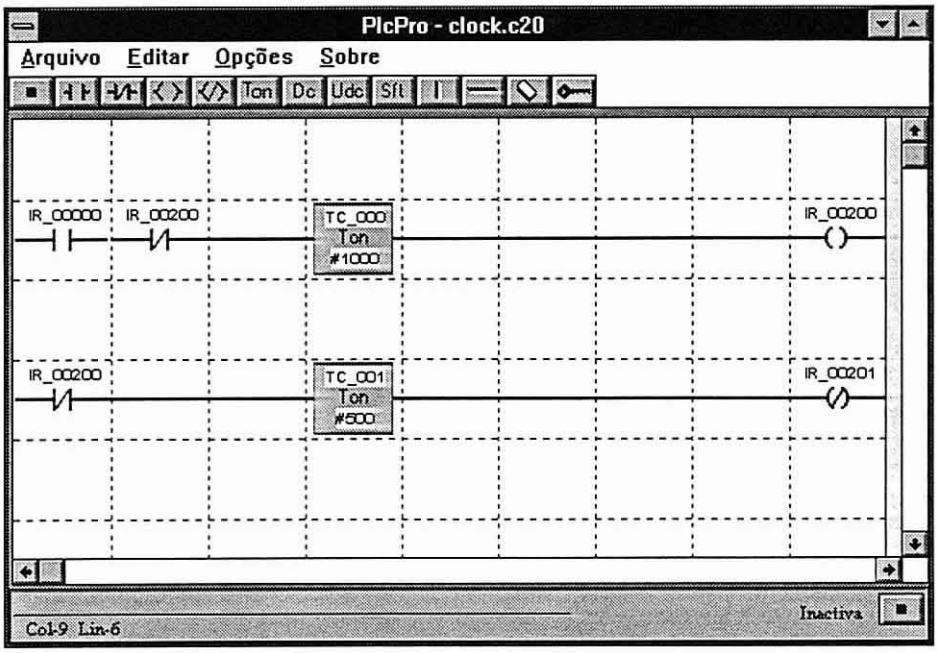


Fig. A.18 - Editor de simulação (programa *clock*)

- Usar o sub-menu Novo do menu Arquivo para criar um novo programa
- Seleccionar a opção  e premir o botão do rato na célula (1,2), isto é, coluna 1-linha 2
- Seleccionar a opção  (para bloquear o mecanismo de opções)
- Seleccionar a opção  e premir o botão do rato nas células (2,2) e (1,4)
- Seleccionar a opção  e premir o botão do rato na célula (9,2)
- Proceder do mesmo modo, para inserir todos os outros símbolos, incluindo as ligações horizontais.

Em seguida, há que editar os símbolos para se inserirem os respectivos parâmetros:

- Seleccionar a opção 
- Premir duas vezes o botão do rato sobre a célula (4,2)
- Preencher a caixa de diálogo da seguinte forma:

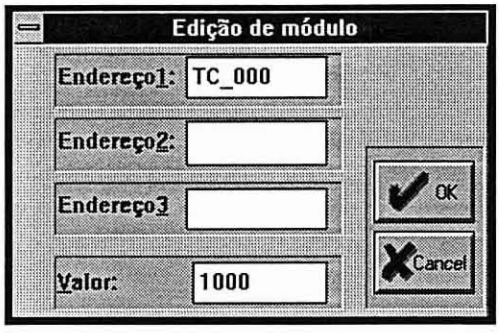
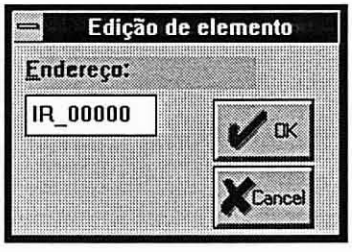


Fig. A.19 - Edição de módulo (programa *clock*)

- Premir duas vezes o botão do rato sobre a célula (1,2)
- Preencher a caixa de diálogo:



Para os outros símbolos procede-se de forma idêntica.

Finalmente, deve-se gravar o programa, por exemplo sob o nome *clock*, usando o sub-menu Gravar (ou Gravar Como) do menu Arquivo.

A.6.2 Configuração dos parâmetros de simulação

Os parâmetros para simulação devem ser inicializados com os valores indicados na janela de informações do editor de estímulos (figura A.20). Para isso basta seleccionar, através dos menus convenientes, o período de 50ms, o tempo de simulação de 20 segundos e o factor de ampliação de 2.

A.6.3 Definição dos estímulos de entrada

No exemplo proposto existe um só estímulo de entrada (Início), cujo endereço é *IR_00000*. Para editar o endereço, deve-se premir o botão do rato na posição correspondente e escrever o endereço. Para definir o diagrama temporal do estímulo utiliza-se o rato, como se fosse um lápis, desenhando o diagrama apresentado na figura A.20.

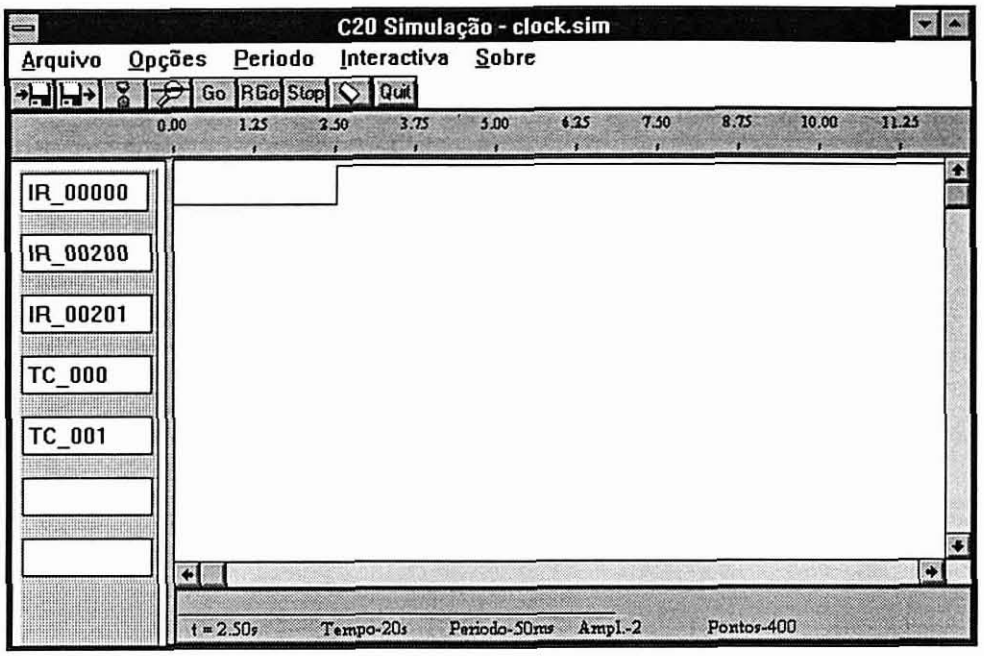


Fig. A.20 - Editor de estímulos (programa *clock* - estímulo de entrada de Início - *IR_00000*)

A.6.4 Definição das saídas e pontos internos

Neste exemplo há duas saídas (*IR_00200* e *IR_00201*) e dois pontos internos (*TC_000* e *TC_001*, que são os *bits* de saída dos temporizadores). Deve-se escrever, nas posições indicadas na área de endereços, os endereços *IR_00200*, *IR_00201*, *TC_000* e *TC_001*, por forma a visualizar os resultados da simulação nessas saídas e nesses pontos internos.

A.6.5 Simulação do programa

A.6.5.1 Simulação em tempo real

Executando a simulação em tempo real (opção RGo), verifica-se que a saída 1 (correspondente a *IR_00201*) do PLC virtual fica activa de forma intermitente com intervalos de 0.5 segundo.

A.6.5.2 Simulação rápida (usando o editor de estímulos)

Com a simulação rápida (opção Go) podem avaliar-se os resultados usando o editor de estímulos. Para uma melhor análise temporal deve-se seleccionar o sub-menu Linha no rato do menu Opções, nomeadamente pode verificar-se que a saída 1 fica activa, pela primeira vez, um segundo depois do sinal Início (*IR_00000*) ficar activo.

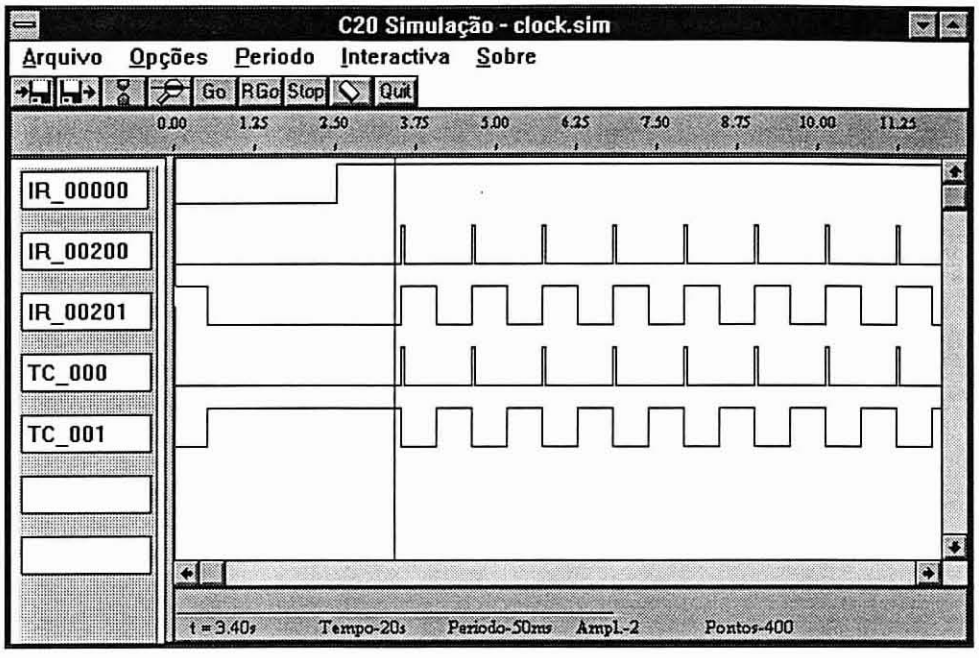


Fig. A.21 - Janela de estímulos (programa *clock* - resultados da simulação)

A.6.5.3 Simulação interactiva

É possível executar a simulação interactiva de tal modo que o sinal de Início seja introduzido pelo próprio operador durante a simulação. Seleccione o sub-menu Simul_Int do menu Interactiva.

É necessário, agora, construir a janela interactiva, que poderá ter o aspecto da figura A.22. Para isso deve criar-se 1 objecto do tipo entrada e 3 objectos do tipo saída/ponto interno:

- Seleccione uma vez a opção Inp, e três vezes a opção Out.

Em seguida, deve editar o endereço e a etiqueta de cada objecto. Por exemplo, para o objecto do tipo entrada, prima o botão do rato duas vezes sobre este, e preencha a respectiva caixa de diálogo (ver figura A.23).

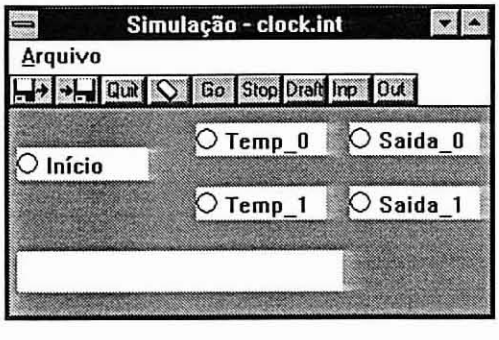


Fig. A.22 - Janela interactiva para o programa *clock*

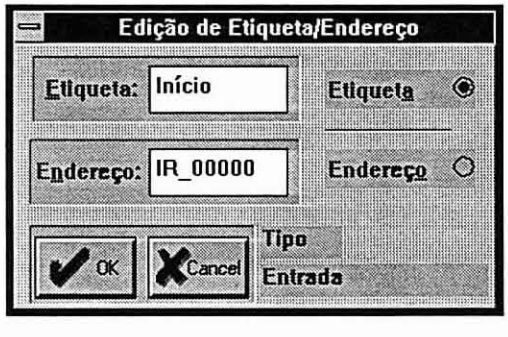


Fig. A.23 - Edição dos parâmetros para a entrada

Proceder da mesma forma para os objectos do tipo saída/ponto interno.

Finalmente, deve deslocar os objectos para as posições indicadas na figura A.22 (isto se for esse o aspecto desejado) e, em seguida, deve gravar a janela de estímulos por exemplo sob o nome de *clock*.

Usando a opção Go, verifique, por exemplo, que a saída 1 "pisca" uma vez no início, e só volta a "piscar" novamente quando premir com o rato sobre a entrada Início. A figura A.24 mostra uma simulação interactiva a correr, muito embora a imagem seja pouco expressiva, porque parada.

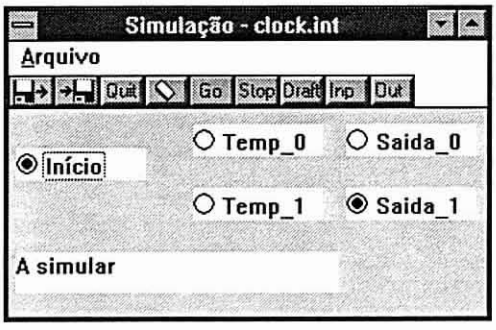


Fig. A.24 - Janela interactiva em simulação