UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO
DEPARTAMENTO DE ELETRÔNICA E MICROELETRÔNICA

Uma Proposta de Linguagem Visual Orientada a Objetos para Programação de Microcontroladores

Antonio Heronaldo de Sousa

Orientador: Prof. Dr. Elnatan Chagas Ferreira

Comissão Julgadora:

Elnatan Chagas Ferreira - FEEC/UNICAMP - Presidente
Oséas Valente de Avilez Filho - FEEC/UNICAMP
Beatriz M. Daltrini – FEEC/UNICAMP
José Carlos Maldonado – USP/São Carlos
Frank H. Behrens – PUC/Campinas

Tese apresentada à Faculdade de Engenharia Elétrica da Universidade Estadual de Campinas, como parte dos requisitos exigidos para a obtenção do título de Doutor em Engenharia Elétrica.

Fevereiro – 1999

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

So85p

Sousa, Antonio Heronaldo de
Uma proposta de linguagem visual orientada a objetos para programação de microcontroladores. / Antonio Heronaldo de Sousa.--Campinas, SP: [s.n.], 1999.

Orientador: Elnatan Chagas Ferreira.
Tese (doutorado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Programação visual (Computação). 2. Linguagem orientada a objetos (Computação). 3. Sistemas embutidos de computador. I. Ferreira, Elnatan Chagas. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

À minha esposa Cláudia e aos meus filhos Vinícius e Cecília,
pela alegria de viver que eles me proporcionam.

Agradecimentos:

Ao Prof. Elnatan, pela orientação e constante incentivo.

Ao Prof. Oséas, que me proporcionou a entrada na UNICAMP e por sugestões ao trabalho.

Ao Dep. de Eng. Elétrica da CCT-UDESC, pela confiança em mim depositada.

À CAPES, pelo apoio financeiro.

Às demais pessoas que contribuíram para a realização deste trabalho.

RESUMO

Este trabalho apresenta um estudo sobre a viabilidade de se utilizar novas metodologias de engenharia de *software* em ambientes microcontrolados. É abordado o uso da programação orientada a objetos, juntamente com a programação visual, na construção de linguagens mais acessíveis e mais produtivas para a programação de microcontroladores. Como resultado, foi proposto um protótipo de uma linguagem, chamada O++. Esta linguagem se caracteriza por utilizar, em conjunto, estruturas gráficas e estruturas textuais para melhor representar dados e algoritmos. Além disso, ela foi projetada para combinar as características de *reusabilidade* da programação orientada a objetos e a *acessibilidade* da programação visual. Ela permite o desenvolvimento de aplicações usando estruturas visuais orientadas a objetos, a fim de melhorar a qualidade e acessibilidade de informações trocadas no desenvolvimento de *software* para sistemas microcontrolados.

ABSTRACT

This thesis presents a feasibility investigation about use of new engineering software methodologies in microcontroller systems. The approach is to use object oriented programming paradigm with visual programming to build a more productive microcontroller programming language.

It was developed a microcontroller programming language, called O++. The main feature of this language is to use, at the same time, graphical structures and text to represent data and procedures. Also, it was designed to combine the reusability of object-oriented programming and the accessibility of visual programming. It allows the development of applications using visual object-oriented structures in order to improve quality and accessibility of information exchange in microcontroller systems software design.

ÍNDICE

INTRODUÇÃO GERAL

Com o aumento da complexidade dos produtos baseados em microcontroladores, que notadamente possuem um *software* implementado em um grande número de linhas de complicado código, criou-se a necessidade do uso de novas metodologias de engenharia de *software*, como a *programação orientada a objetos* (Object-Oriented Programming) e a *programação visual* (Visual Programming).

Esses produtos, na grande maioria, são de difícil manutenção, principalmente por serem implementados de forma puramente textual. Assim sendo, novas metodologia, tais como a programação orientada a objetos e a programação visual, podem facilitar tanto o desenvolvimento como a manutenção, haja visto que os problemas inerentes à programação textual [SOU95] são, por exemplo, minimizados pelo uso de uma linguagem visual (ou linguagem gráfica) e uma maior reusabilidade é permitida pelo uso da programação orientada a objetos [WIN91].

A linguagem O++, proposta neste trabalho, pretende combinar as vantagens de cada enfoque: a reusabilidade e extensibilidade da tecnologia orientada a objetos e a acessibilidade da programação visual. A linguagem foi proposta incorporando estruturas visuais orientadas a objetos para microcontroladores, permitindo melhorar a qualidade e acessibilidade de informações trocadas no desenvolvimento de *software* para sistemas microcontrolados.

Principais contribuições

Esta tese apresenta duas importantes contribuições. A primeira se refere ao estudo realizado sobre a viabilidade de se utilizar novas tecnologias de *software* para a programação de microcontroladores, enfocando as deficiências das linguagens tradicionais em responder aos problemas dessa área e apontando uma alternativa com o uso da programação orientada a objetos e da programação visual. A segunda, coroando a primeira, foi o desenvolvimento de um protótipo de um ambiente visual orientado a objetos para a programação de sistemas microcontrolados.

Panorama geral da tese

No capítulo 1, são feitas algumas **considerações sobre os microcontroladores**, com o intuito de fornecer uma visão sobre suas aplicações, sua evolução, fabricantes que dominam o mercado, principais características técnicas, linguagens de programação e ferramentas de desenvolvimento de *software*. Com isso, objetiva-se mostrar a real necessidade de se buscar novas alternativas para tornar compatível o crescimento do uso dos microcontroladores com a produtividade de suas ferramentas de *software*.

O capítulo 2 dá sustentação teórica para a premissa de que **o uso de linguagens visuais pode melhorar a produtividade dos desenvolvedores de *software***. No item 2.1, enfoca-se a evolução dos ambientes de programação textual e os problemas, aos quais estão sujeitas, as aplicações desenvolvidas nesse tipo de ambiente. No item 2.2, a programação visual é abordada como alternativa para resolver alguns problemas dos ambientes textuais. Relata-se algumas teorias que apoiam o uso de linguagens visuais e as implicações que podem ocorrer com o seu uso.

O capítulo 3 fornece os elementos teóricos para a segunda premissa, **o uso da programação orientada a objetos pode, também, melhorar a produtividade dos desenvolvedores de *software***. Neste capítulo é feito um apanhado histórico da evolução do *software*, apontando os aspectos que fizeram uma convergência para a engenharia de *software* e para a programação orientada a objetos.

No capítulo 4, é feita **uma proposta de uma linguagem visual orientada a objetos para programação de microcontroladores**, batizada de O++. Esta proposta une os paradigmas da programação orientada a objetos e da programação visual. O capítulo, inicialmente, fornece uma visão de como essa união pode gerar resultados positivos para o desenvolvimento de *software*. Também é fornecida uma visão da estrutura da linguagem proposta, ressaltando os aspectos de filosofia de programação e detalhes psicológicos levados em consideração em seu projeto.

O capítulo 5 mostra **os resultados obtidos** no trabalho, relatando-se o desenvolvimento do protótipo O++. Primeiro, é feita uma descrição do ambiente O++ do ponto de vista operacional, enfocando-se os aspectos de utilização do programa. É dada uma descrição dos mecanismos utilizados na interface com o usuário, bem como das operações implementadas especificamente para microcontroladores. Num item seguinte, o

foco é a implementação do compilador. É abordado o modelamento da linguagem, a estrutura do código e os mecanismos de tradução usados, a geração do código em linguagem C e o mecanismo utilizado para implementar a biblioteca de funções.

Finalmente, são relatadas as conclusões do trabalho e sugeridas as propostas para trabalhos futuros.

CAPÍTULO 1

CONSIDERAÇÕES SOBRE OS MICROCONTROLADORES

Introdução

Neste capítulo são feitas algumas considerações sobre os microcontroladores, com o intuito de fornecer uma visão sobre suas aplicações, sua evolução, fabricantes que dominam o mercado, principais características técnicas, linguagens de programação e ferramentas de desenvolvimento de *software*. Com isso, objetiva-se mostrar a real necessidade de se buscar novas alternativas para tornar compatível o crescimento do uso dos microcontroladores com a produtividade de suas ferramentas de *software*.

1.1. Considerações iniciais

Na área tecnológica é grande o fascínio por trabalhar com temas que imediatamente produzam, ou a curto prazo possam produzir, resultados diretamente ligados ao aumento de produtividade e/ou diminuição de custos. Evidentemente, esses trabalhos são incentivados por órgãos ligados à indústria, que devido à dinâmica do mercado, precisa oferecer respostas rápidas às suas necessidades. Infelizmente, essas respostas nem sempre se traduzem em um bem comum. Atualmente estamos presenciando uma verdadeira ressaca tecnológica, onde a automação, principalmente a robótica, tem provocado uma grande diminuição nos quadros de mão-de-obra.

As transformações que estamos presenciando são, sem dúvida, alimentadas pelos computadores, que para a grande maioria das pessoas são traduzidos pelos PCs e *mainframes*. Muitas dessas pessoas não sabem, mas um grande número de pequenos processadores (microcontroladores), embutidos em diversos equipamentos, exercem um papel importante no dia a dia delas. Despertar ao som de um CD *Player* programável, tomar café da manhã preparado por um microondas digital, e ir ao trabalho de carro, cuja injeção de combustível é microcontrolada, são apenas alguns exemplos. Paradoxalmente, para muitas dessas pessoas, "os computadores são apenas máquinas feitas para complicar as coisas".

Os microcontroladores estão presentes também no dia a dia de pessoas de classes economicamente inferiores. Operários registram seu horário de entrada e saída no trabalho através de relógio-ponto microcontrolado e vão ao trabalho em transporte coletivo, cujo sistema de registro de passageiros é automatizado.

1.2. Os microcontroladores e suas aplicações

O mercado para microcontroladores continua em franca expansão, ampliando seu alcance principalmente em aplicações residenciais, industriais, automotivas e de telecomunicações. Segundo dados da National Semiconductor [NAT97], uma residência típica americana possui 35 produtos baseados em microcontrolador. A estimativa é de que até o início do próximo milênio, aproximadamente 250 produtos residenciais sejam comandados por esse tipo de dispositivo.

Num passado recente, o alto custo dos dispositivos eletrônicos limitou o uso dos microcontroladores apenas aos produtos domésticos considerados de alta tecnologia (tv, vídeo e som). Porém, com a constante queda nos preços dos *chips*, os microcontroladores passaram a ser utilizados em produtos menos sofisticados do ponto de vista da tecnologia, como máquinas de lavar, microondas, fogões e refrigeradores. Assim, a introdução do microcontrolador nestes produtos cria uma diferenciação e permite a inclusão de melhorias de segurança e de funcionalidade. Alguns mercados chegaram ao ponto de tornar obrigatório o uso de microcontroladores em determinados tipos de equipamentos, impondo um pré-requisito tecnológico.

Muitos produtos que temos disponíveis hoje em dia, simplesmente não existiriam, ou não teriam as mesmas funcionalidades sem um microcontrolador. É o caso, por exemplo, de vários instrumentos biomédicos, instrumentos de navegação por satélites, detetores de radar, equipamentos de áudio e vídeo, eletrodomésticos, dentre outros.

Entretanto, o alcance dos microcontroladores vai além de oferecer algumas facilidades. Uma aplicação crucial, onde os microcontroladores são utilizados, é na redução de consumo de recursos naturais. Existem sistemas de aquecimento modernos que captam a luz solar e, de acordo com a demanda dos usuários, controlam a temperatura de forma a minimizar perdas. Um outro exemplo, de maior impacto, é o uso de microcontroladores na

redução do consumo de energia em motores elétricos, que são responsáveis pelo consumo de, aproximadamente, 50% de toda eletricidade produzida no planeta[GAN92]. Portanto, o alcance dessa tecnologia tem influência muito mais importante em nossas vidas, do que se possa imaginar.

O universo de aplicações dos microcontroladores, como já mencionado, está em grande expansão, sendo que a maior parcela dessas aplicações é em sistemas embutidos. A expressão "sistema embutido" se refere ao fato do microcontrolador ser inserido nas aplicações (produtos) e usado de forma exclusiva por elas. Como a complexidade desses sistemas cresce vertiginosamente, o *software* tem sido fundamental para oferecer as respostas às necessidades desse mercado. Tanto é, que o *software* para microcontroladores representa uma fatia considerável do mercado de *software* americano. Segundo Edward Yourdon [YOU91] [YOU97] a proliferação dos sistemas embutidos, juntamente com o advento da Microsoft, são os responsáveis pela retomada do crescimento da indústria de *software* naquele país.

1.3. Evolução dos microcontroladores

Antes dos microcontroladores, os sistemas de controle eram construídos, exclusivamente, por componentes lógicos discretos, e portanto, apresentavam grandes dimensões físicas. Posteriormente, os microprocessadores foram usados para implementar esses sistemas, o que permitiu grande redução no número de componentes utilizados. Até hoje é comum encontrarmos sistemas à base do Zilog Z80, Intel 8088, Motorola 6809 e outros.

Com a continuação do processo de miniaturização, todos os componentes necessários para um controlador foram integrados num único *chip*. Daí surgiu o termo microcontrolador. Um microcontrolador pode incorporar todas, ou quase todas, as partes necessárias para implementar um controlador.

Usualmente, os microcontroladores incorporam uma CPU, memória RAM, algum tipo de memória ROM, porta serial, portas paralelas de entrada/saída, contadores e temporizadores, e um controlador de interrupções. Além disso, eles possuem um amplo conjunto de instruções para manipular *bits*, realizar acesso direto aos pinos de entrada/saída,

e um rápido e eficiente sistemas de processamento de interrupções. Com isso, e aliada à drástica redução do número de componentes envolvidos e a uma maior simplicidade de projeto, os microcontroladores se tornaram um grande atrativo para implementar sistemas de controle a um custo relativamente baixo.

Os microcontroladores são freqüentemente utilizados para controlar equipamentos das mais variadas áreas: eletrodomésticos, equipamentos automotivos, controle ambiental, indústria bélica e aeroespacial, instrumentação, controle de processos, automação industrial e muitas outras. A complexidade dos microcontroladores varia de acordo com a aplicação. Há microcontroladores bem simples usados para implementar pequenos sistemas (uma calculadora, por exemplo), como também há microcontroladores mais complexos usados para implementar sistemas maiores (sistema de comando por voz, por exemplo).

Na robótica é comum encontrar vários microcontroladores trabalhando em conjunto. Cada microcontrolador é responsável por uma parte do sistema e, através de um sistema de comunicação, se comunica com uma unidade central, de maior porte, repassando informações coletadas e recebendo novos parâmetros de controle.

Uma aplicação em especial, os sistemas coletores de dados, evidencia as principais vantagens do uso de microcontroladores: dimensões físicas reduzidas, baixo consumo e flexibilidade. Sem essas características seria muito difícil realizar a coleta automatizada de dados no chão de fábrica e o monitoramento e armazenagem de dados climatológicos em um balão, só para citar alguns exemplos.

O processo de evolução dos microcontroladores pode ser dividido em três fases distintas. Primeiro sugiram os microcontroladores baseados em uma tecnologia de *hardware* relativamente primitiva, com grande limitação de memória e de processamento. Esses sistemas são incapazes de armazenar outros dados além dos imprescindíveis ao processamento. Além disso, são usados em aplicações cuja interface com o usuário é precária (implementada com LEDs ou *displays* de sete segmentos), e não permitem a comunicação com outros sistemas. Eles operam independentemente, isolados dos demais.

Na segunda fase, o *hardware* apresenta complexidade bem maior do que o da geração anterior. As interfaces com o usuário utilizam *displays* de cristal líquido, tornando o processo de comunicação mais amigável. Também é possível a comunicação com outros

sistemas, através de interfaces seriais de comunicação. Os custos com memória e mais processamento já não são tão significativos. Entretanto, o *software* se torna uma parte preponderante da aplicação, por causa da interface mais eficiente com o usuário e de uma maior funcionalidade do sistema.

A última fase, ainda emergindo, incorpora as características de *hardware* e *software* dos sistemas *desktops* atuais. Processadores de 32 *bits*, alta capacidade de memória, interfaces de comunicação com redes heterogêneas e uma grande variedade de recursos para processamento digital de sinais e outras tarefas.

1.4. Fabricantes de microcontroladores

O mercado de microcontroladores é, atualmente, dominado pelos tradicionais fabricantes de microprocessadores, a Intel e a Motorola. Entretanto, há quase duas dezenas de outros fabricantes que possuem, juntos, uma fatia considerável do mercado. O anexo I mostra uma tabela com os principais fabricantes de microcontroladores que atuam no mercado. Esta tabela relaciona as principais famílias com suas características mais comuns, oferendo um guia rápido sobre as pontencialidades dos microcontroladores disponíveis atualmente no mercado. A tabela foi construída, principalmente, a partir de dados coletados diretamente nos *sites* dos fabricantes, via rede INTERNET. Também foram consultados manuais técnicos dos próprios fabricantes.

Além da Intel e Motorola, destacam-se a Microchip, Texas, National, Hitachi, Sharp e Zilog.

- Intel Corporation

O microcontrolador mais popular da Intel é o 8051, que representa a sua segunda geração de microcontroladores. Ele substituiu o 8048, que foi o primeiro microcontrolador de propósito geral lançado no mercado. O 8051 lidera o mercado de microcontroladores no momento, sendo fabricado também pela Philips e Siemens. Sua arquitetura é baseada na separação de memória de dados e programa. No total, a família MCS-51 endereça até 128K *bytes*, 64K para código e mais 64K para dados. Além disso, possui até 256 *bytes* de

memória RAM interna, mais um conjunto de registros de funções especiais (SFRs). Possui E/S mapeada em seu próprio espaço, dispondo de quatro portas de E/S.

O 8051 é chamado "processador boleano ", pois possui um vasto repertório de instruções para manipular *bits*. Ele é considerado de fácil programação, sendo o microcontrolador com maior número de ferramentas de desenvolvimento disponíveis.

A terceira geração iniciou com o 8096, que trabalha com 16 *bits*. Este microcontrolador possui uma maior quantidade de recursos do que os anteriores, dentre as quais podemos destacar: melhor repertório de instruções, um número maior de *bits* de E/S e maior capacidade de memória.

- Motorola Microcontroller Tech. Group

A Motorola fabrica um dos mais populares microcontroladores de 8 *bits*, o 68HC11, sendo considerado o concorrente nato do MCS-51. O 68HC11 herdou o conjunto de instruções dos microprocessadores Motorola 68xx e possui uma arquitetura de memória semelhante ao do 68xx, onde código, dados e dispositivos de E/S compartilham o mesmo espaço de endereçamento.

Na linha de 16 *bits*, destaca-se a série MC68302, que incorpora vários periféricos da família do 68000. A série MC68302 é chamada de processador integrado, pois se constitui numa espécie de super microcontrolador. Se caracteriza por apresentar alta velocidade de processamento e grande capacidade de endereçamento de memória externa, se equiparando a um processador Intel 80386.

- Microchip Technology Inc.

A linha PIC de microcontroladores se tornou bastante conhecida a partir dos anos 80. Seu fabricante, a Microchip desenvolveu seus microcontroladores dentro de um universo de produtos que combina potencialidades com custo mínimo. Os microcontroladores PIC foram os primeiros com arquitetura RISC, cuja simplicidade de projeto permite que mais facilidades sejam adicionadas a um baixo custo. Embora possuindo um pequeno repertório

de instruções (33 instruções para a linha 16C5C, contra 90 do Intel 8048, por exemplo), a linha PIC tem boa variedade de facilidades incluídas como parte do circuito integrado.

Os barramentos separados de dados e instruções (arquitetura baseada no modelo Harvard) permitem acesso simultâneo ao programa e aos dados, sobrepondo algumas operações para aumentar a capacidade de processamento. Os benefícios da simplicidade de projeto se refletem em um *chip* muito pequeno, com um número reduzido de pinos, e baixíssimo consumo. Por essas razões vem dominando uma fatia importante do mercado, a de pequenas aplicações.

- Texas Instruments

O ramo do mercado que a Texas Instruments detem uma fatia considerável é o de DSPs. Na área de microcontroladores, a família mais conhecida é a TMS370. Ela não apresenta grandes novidades em relação às outras famílias de 8 *bits*, entretanto apresenta boa capacidade de endereçamento de memória externa.

- National Semiconductor

A National Semiconductor é conhecida, no mercado de microcontroladores, pela fabricação da família COP8, apesar de oferecer também microcontroladores e 4 e 16 *bits*. A família COP8 apresenta baixo custo e um ótimo conjunto de facilidades. Sua arquitetura de 8 *bits* é baseada no modelo Harvard. O COP8 contém todo sistema de temporização, lógica de interrupção, ROM, RAM e E/S necessários para implementar funções de controle dedicado em uma variedade de aplicações. No entanto, apresenta um conjunto de instruções limitado e apenas três modos diferentes de endereçamento.

- Hitachi America Ltd.

A Hitachi está presente no mercado com microcontroladores de 8, 16 e 32 *bits*. O principal destaque dos microcontroladores da Hitachi é a grande capacidade de memória interna e espaço de endereçamento de memória externa. A família de 8 *bits* H8/300 pode ter

até 4K *bytes* de memória interna. A família de 16 *bits* H85/2000 endereça até 16 mega *bytes* de memória externa, enquanto a família de 32 *bits* SuperH, de tecnologia RISC, pode endereçar até 4 giga *bytes* de memória externa.

- Sharp Microeletronics Technology

A Sharp Microeletronics produz microcontroladores de 8, 16 e 32 *bits*. Seus microcontroladores não apresentam grande diferencial em relação aos demais fabricantes. Sua linha de 32 possui controlador de LCD incorporado ao *chip* do microcontrolador.

- Zilog Inc.

A Zilog, muito conhecida pelo seu memorável Z80, está no mercado de microcontroladores com a família Z8. Esta família é uma recompilação do Z80 com seus periféricos. Porém, novas facilidades foram acrescentadas, como canais de DMA, contadores/temporizadores, E/S com até 32 linhas e um eficiente sistema de interrupção com 6 fontes possíveis. Apesar de apresentar um bom conjunto de instruções, domina uma fatia muito pequena do mercado.

1.5. Características dos microcontroladores

Como podemos observar no anexo I, a maioria dos microcontroladores ainda manipula informações de 8 *bits*. Porém nos últimos anos o mercado vêm convergindo para microcontroladores com maior capacidade de processamento. Muitos fabricantes dispõem de famílias de 16 *bits*, e algumas já oferecem versões de 32 *bits*.

É difícil estabelecer um conjunto completo de características para os microcontroladores, pois o número de fabricantes e de famílias é bastante grande. Assim, será relacionado, a seguir, um conjunto básico de características dos microcontroladores, abordando os aspectos mais comuns desses dispositivos.

- Tecnologia de fabricação

A tecnologia de fabricação de microcontroladores mais comum é a CMOS - Complementary Metal Oxide Semiconductor. Como se sabe, esta tecnologia requer menor consumo que as demais existentes, o que facilita as aplicações em campo, onde o microcontrolador é, normalmente, alimentado por baterias. Além disso, a tecnologia CMOS possui um processo de fabricação mais simples e uma maior densidade de integração, pois o transistor CMOS é menor que o transistor bipolar e as memórias e portas lógicas são construídas apenas com transistores, sem diodos ou resistores.

Como a imunidade a ruído é função dos níveis de tensão utilizados numa tecnologia e como o CMOS permite trabalhar com tensões mais altas, isto torna os níveis lógicos mais definidos e imunes a pequenas variações na fonte de alimentação. Assim, os *chips* CMOS possuem maior imunidade à variação de tensão que as tecnologias anteriores. Porém, em virtude das capacitâncias do modelo CMOS, os efeitos de interferência eletromagnética são maiores nessa tecnologia. Alguns fabricantes incorporam à pastilha do microcontrolador um circuito que filtra esse tipo de ruído.

- Capacidade de Memória

Em termos de memória os microcontroladores apresentam grandes variações, até mesmo dentro de uma mesma família. No caso de memória interna de programa, geralmente, podem chegar a 64K*bytes*. Para a memória interna de dados é comum de 128 a 256 *bytes*, entretanto há famílias com até alguns kilo *bytes*. Em se tratando de endereçamento para memória externa, o montante de memória pode ir de alguns kilo *bytes*, até alguns mega *bytes*. Todas as famílias dispõem de memórias ROM/PROM/EPROM e boa parte delas também dispõe de versões com EEPROM e *flash* EPROM.

- Velocidade de Processamento

A freqüência de trabalho também varia muito de família para família. Temos, por um lado, famílias com freqüência de no máxima 3MHz e, por outro, famílias com freqüência máxima de 60MHz. Porém, na média, os microcontroladores trabalham de 8 a 12MHz.

- Portas de E/S

Todos os microcontroladores possuem portas paralelas de E/S. O número de *bits* varia de algumas unidades a algumas dezenas de *bits*. Estes *bits* podem ser programados para entrada ou saída, ou ainda, trabalhar em modo bidirecional. A maioria dos microcontroladores também possuem uma ou mais interfaces de comunicação serial, cujos parâmetros podem ser configurados via *software*. Alguns deles possuem uma interface compatível I2C (Inter-Integrated Circuit) da Intel. Esta interface especial usa duas linhas para implementar uma rede multi-master e/ou multi-slave com detecção de colisão. Cada nó da rede possui um endereço exclusivo, o qual é introduzido nas mensagens, permitindo que vários dispositivos compartilhem o mesmo meio físico.

- Contadores e Temporizadores

Quase todos os microcontroladores incorporam pelo menos um contador/temporizador programável, usado para as operações de contagem de eventos, geração de pulsos e medição de tempo e de freqüência.

- Conversores Analógico-Digital e Digital-Analógico

Alguns microcontroladores possuem conversores analógico-digital e digital-analógico para implementar a interface com o mundo analógico (tipicamente um sinal em tensão). Esses microcontroladores são muitos utilizados em aplicações que envolvem aquisição de dados e/ou controle.

- Comparadores Analógicos

Alguns microcontroladores dispõem de um ou mais comparadores analógicos padrões. Estes comparadores podem ser utilizados para várias funções, como detecção de curto-circuito, conversão analógico-digital, detecção de limiar de mudança de nível, dentre outros.

- Gerador de PWM (Pulse Width Modulator)

Muito usado em microcontroladores como técnica alternativa de conversão digital-analógica, onde um trem de pulsos é gerado com largura variável em função de alguma propriedade de saída e, posteriormente, regulado por um filtro passa-baixa. A saída do filtro é uma tensão proporcional ao *duty cycle* do trem de pulso.

- Modo de Baixo Consumo

Muitos microcontroladores podem trabalhar em modo de baixo consumo de energia (Idle/Halt/Wakeup), o que é apropriado para aplicações em campo. Nessas aplicações, muitas vezes o microcontrolador fica ativo apenas quando um determinado evento ocorre, e tão logo seja gerada uma resposta, o microcontrolador volta ao estado inativo. O consumo de corrente elétrica, neste estado, é, tipicamente, de 1μA.

- Sistema de Rearme Automático (Watchdog)

O sistema de rearme automático permite que o microcontrolador seja reiniciado, caso ocorra algum problema no *software* ou no *hardware*. Basicamente, o dispositivo de *watchdog* fica esperando um pulso a cada intervalo pré-programado de tempo. Caso o programa falhe, não enviando o pulso dentro do intervalo estabelecido, o microcontrolador é reiniciado. O problema pode não ser resolvido, mas operações indesejáveis são assim evitadas pela reiniciação do sistema.

- Monitor de Clock

Este é um dispositivo de segurança habilitado por *software*, que permite paralisar o microcontrolador (deixá-lo em permanente estado de reset), caso a freqüência de trabalho caia abaixo de um certo patamar. Após a normalização da freqüência de trabalho, o microcontrolador volta a trabalhar em estado de execução.

- Proteção à baixa tensão (Brownout protection)

Este é um outro dispositivo de segurança habilitado por *software*, que permite paralisar o microcontrolador no caso da tensão de alimentação ficar abaixo de um certo limiar. Após a normalização da tensão de alimentação, o microcontrolador volta a trabalhar em estado de execução.

- Sistema de Proteção ao *Software*

Alguns microcontroladores vêm com algum tipo de dispositivo de proteção ao *software*. Estes dispositivos, normalmente, são de dois tipos: os que trabalham com criptografia e os que trabalham com "detonadores", que são capazes de destruir o software. A finalidade é proteger o programa armazenado no microcontrolador contra intromissões não autorizadas (engenharia reversa, modificações, pirataria, etc).

- Monitor Residente

Um monitor é um programa instalado na memória de programa do microcontrolador, com a finalidade de prover capacidades básicas de desenvolvimento. Dentre estas se incluem: a partir de um terminal, realizar carga de programa e/ou dados para a memória do microcontrolador, via interface serial; executar programas; examinar e modificar memória e registradores; estabelecer *break-points*; e outras funções de depuração.

- Interpretador BASIC Residente

Alguns microcontroladores incorporam em sua memória de programa, um interpretador BASIC, o que permite que, a partir de um terminal, possa ser desenvolvido uma aplicação usando a linguagem BASIC, reduzindo os custos com ferramentas de desenvolvimento.

1.6. Linguagens de programação para microcontroladores

O desenvolvimento de *software* para sistemas microcontrolados não é realizado usando-se apenas os recursos do microcontrolador, que não são muito apropriados. É comum se utilizar outros dispositivos, como por exemplo, os microcomputadores pessoais

que oferecem maior capacidade de recursos (mais memória, *display* gráfico, teclado e *mouse*, memória de massa, dentre outros). Desta forma pode-se utilizar esses recursos para criar um ambiente mais amigável aos programadores de microcontrolador.

Mesmo assim, por ser, inicialmente, um tipo de dispositivo com baixa quantidade de memória e estar sempre empregado em aplicações com forte interação com o *hardware*, os microcontroladores sempre tiveram sua programação associada com a linguagem *Assembly*, como ilustra a figura a seguir.

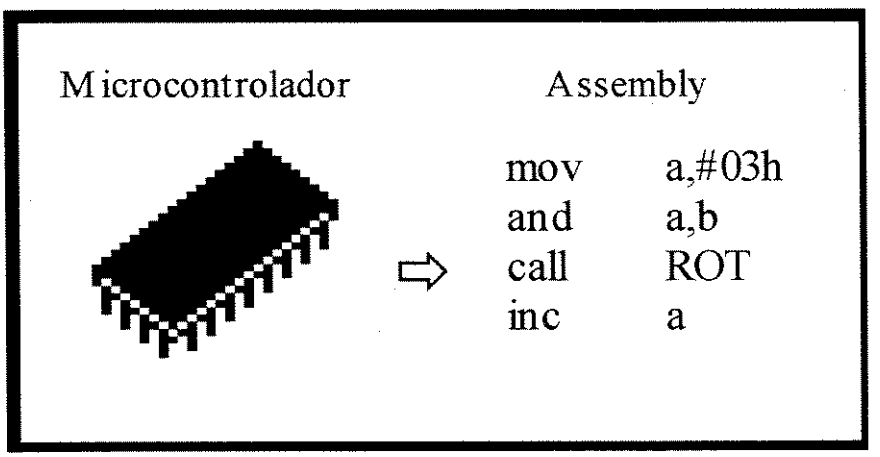

Figura 1.1: A forte ligação do microcontrolador com a linguagem *Assembly*

- Linguagem *Assembly*

A linguagem *Assembly* dos microcontroladores é semelhante à linguagem *Assembly* dos microprocessadores. O conjunto de instruções dispõe de um maior número de operações para manipular *bits*, em comparação aos microprocessadores usuais, entretanto o fato dos microcontroladores terem normalmente diferentes regiões de memória torna as coisas significativamente complicadas. As instruções de movimentação de dados, lógicas e desvio de execução são geralmente similares as da maioria dos outros microprocessadores. Desta forma, para quem já trabalhou com linguagem *Assembly* de qualquer microprocessador, o processo é o mesmo, com suas vantagens e desvantagens.

Além da linguagem *Assembly*, são muito utilizadas na programação de microcontroladores, três outras linguagens: PL/M, BASIC e C. Todas elas consideradas de alto nível, em comparação à linguagem *Assembly*. [SCH93]

- Linguagem PL/M

O PL/M é uma linguagem elaborada pela INTEL e tem sido disponível para seus microprocessadores, começando com o 8080. Ela se assemelha ao PASCAL, mas se originou do PL-1. Como o C, ela é uma linguagem estruturada, mas usa muitos arranjos de palavras chaves para definir suas estruturas. O compilador PL/M produz um código fortemente compacto, tão bom quanto um programa escrito em *Assembly*. O PL/M é muito mais fácil de se usar que a linguagem *Assembly*, pois os compiladores e "linkers" gerenciam detalhes de alocação de variáveis e movimentação de dados entre as áreas de memória. Pode-se dizer que PL/M é uma "linguagem Assembly de alto-nível", tanto no sentido negativo como no positivo, apresentando as mesmas vantagens e desvantagens. Ela permite controlar vários detalhes de geração de código, mas para microcontroladores, PL/M não comporta números complexos, variáveis tipo ponto flutuante, ou funções trigonométricas.

- Linguagem BASIC

Outra linguagem utilizada na programação de sistemas dedicados é o BASIC, que é facilmente encontrada em computadores IBM-PC e é comumente a primeira linguagem de programação que se aprende. Ela atende bem ao seu propósito: o BASIC é uma linguagem de introdução à programação.

O BASIC é muito fácil de se usar. Na maioria das implementações, ela é interpretada, o que possibilita detectar erros ao final de cada linha do programa, ao invés de conhecê-los somente quando o programa termina de ser traduzido. Porém, existem duas razões pelas quais o BASIC não é conveniente em sistemas dedicados.

Em primeiro lugar, como ele é interpretado, ele é naturalmente lento. Cada linha deve ser convertida para o código de máquina toda vez que for executada. O processo de interpretação faz com que seja perdido muito tempo de processamento, que deveria ser usado para a aplicação propriamente dita. Existem versões do BASIC compilado (QuickBASIC, por exemplo), que evitam esse problema. Entretanto, não há até o momento, nenhuma versão comercialmente difundida do BASIC compilado para microcontroladores.

Em segundo lugar, pode-se destacar a inconveniente simplificação no uso de variáveis. Todas as variáveis são, usualmente, implementadas como ponto-flutuante, o que resulta na necessidade de se executar rotinas complexas, mesmo para valores tipo inteiro. Isto torna os programas lentos e grandes.

Pode-se dizer que o BASIC, no contexto de sistemas dedicados, deve ser indicado para aplicações onde a facilidade de programação seja mais importante que a eficiência ou que a velocidade.

- Linguagem C

Finalmente, pode-se destacar a linguagem C, que surgiu há mais de trinta anos com o sistema operacional UNIX. Ela é estruturada e produz um código compacto. A estrutura da linguagem C é marcada pelas chaves "{}" delimitadoras de blocos, ao invés de palavras reservadas (begin...end, por exemplo). A linguagem C faz uso de símbolos especiais que raramente são usados na escrita quotidiana. Ela permite atingir detalhes de controle da máquina sem recorrer ao *Assembly*. Entretanto, os programas em C podem ser tão condensados que sua manutenção fica bastante dificultada.

Desde 1985, o compilador C está disponível para microcontroladores. Há um grande número de fabricantes que oferecem este compilador para quase todas as famílias de microcontroladores. Cada qual com suas vantagens e desvantagens.

No anexo II, há uma relação com a maioria dos fabricantes que, atualmente, oferecem o compilador C para microcontroladores. Eles oferecem código padrão ANSI e geração de código recursivo e reentrante. Alguns deles permitem trabalhar com o conceito de bancos de memória, quebrando a barreira de endereçamento do microcontrolador. Só para citar um exemplo, a versão do Archimedes C para a família MCS-51 da Intel pode trabalhar com até 2 mega *bytes* de memória externa.

O Archimedes e o Franklin são os mais empregados. O Franklin pelo seu código compacto e facilidade de uso; o Archimedes pelas suas facilidades complementares e boa documentação. Após estes, vêm os da Tasking e Avocet. O produto da Tasking é razoavelmente rápido sem geração excessiva de código. Já o compilador Avocet dispõe de

uma excelente documentação. Isso para mencionar apenas os quatro mais conhecidos. [SCH93]

Apesar das diferentes vantagens e desvantagens oferecidas pelas três linguagens, elas têm em comum o fato de o programa-fonte ser expresso por um texto (por razões históricas, um texto em inglês). Desta forma, essas linguagens aproximam o programa escrito para o computador à linguagem humana, umas em maior grau que outras. Entretanto, o nível de detalhamento das instruções do programa sintetizadas em texto, torna sua elaboração e manutenção, em muitos casos, bastante complicadas.

1.7. Programação orientada a objetos usando a linguagem C

No âmbito dos sistemas microcontrolados, podemos destacar duas propostas ([SAM97] e [SIC97]) de implementação dos conceitos da programação orientada a objetos, usando a linguagem C para microcontroladores. As duas utilizam estruturas de dados (ver figura 1.2) semelhantes que permitem implementar encapsulamento, hereditariedade e polimorfismo. Porém, estas estruturas geram código excessivo, aumentando as necessidades de memória e de processamento.

Segundo as duas propostas, para cada objeto é criado um par de apontadores: um para indicar a posição inicial dos membros de dados e outro para indicar a classe a qual o objeto pertence. Cada apontador representa (na maioria dos compiladores C para microcontroladores) um endereço de 16 *bits*, ou seja, cada objeto criado aloca 4 *bytes* extras de memória. São alocados ainda 8 *bytes* extras para cada classe: dois para o apontador da classe pai, dois para o apontador da tabela de mensagens da classe, dois para armazenar o tamanho da área de dados e dois para armazenar o número de mensagens da classe. Além disso, é criada uma tabela de apontadores para os métodos responsáveis pelo atendimento das mensagens, sendo alocados 2 *bytes* extras para cada mensagem.

Outro ponto importante a ser observado nessas propostas, é que há um código extra de iniciação das estruturas de dados, o que pode gerar gargalos de processamento se o número de classes e de instâncias delas não for devidamente controlado.

Além desses inconvenientes, a versão do código orientado a objetos em C é menos legível e de manutenção mais difícil que a versão procedimental, haja visto a sintaxe artificial

de utilização das estruturas auxiliares. Porém, os métodos propostos são consistentes e permitem o reuso de código. A adição de novos membros à uma classe pai não requer qualquer alteração manual nas classes filhas. E, apesar das desvantagens apresentadas, as propostas são válidas, pois o ganho de produtividade obtido pelo reuso de código pode justificar as perdas de memória, velocidade e legibilidade.

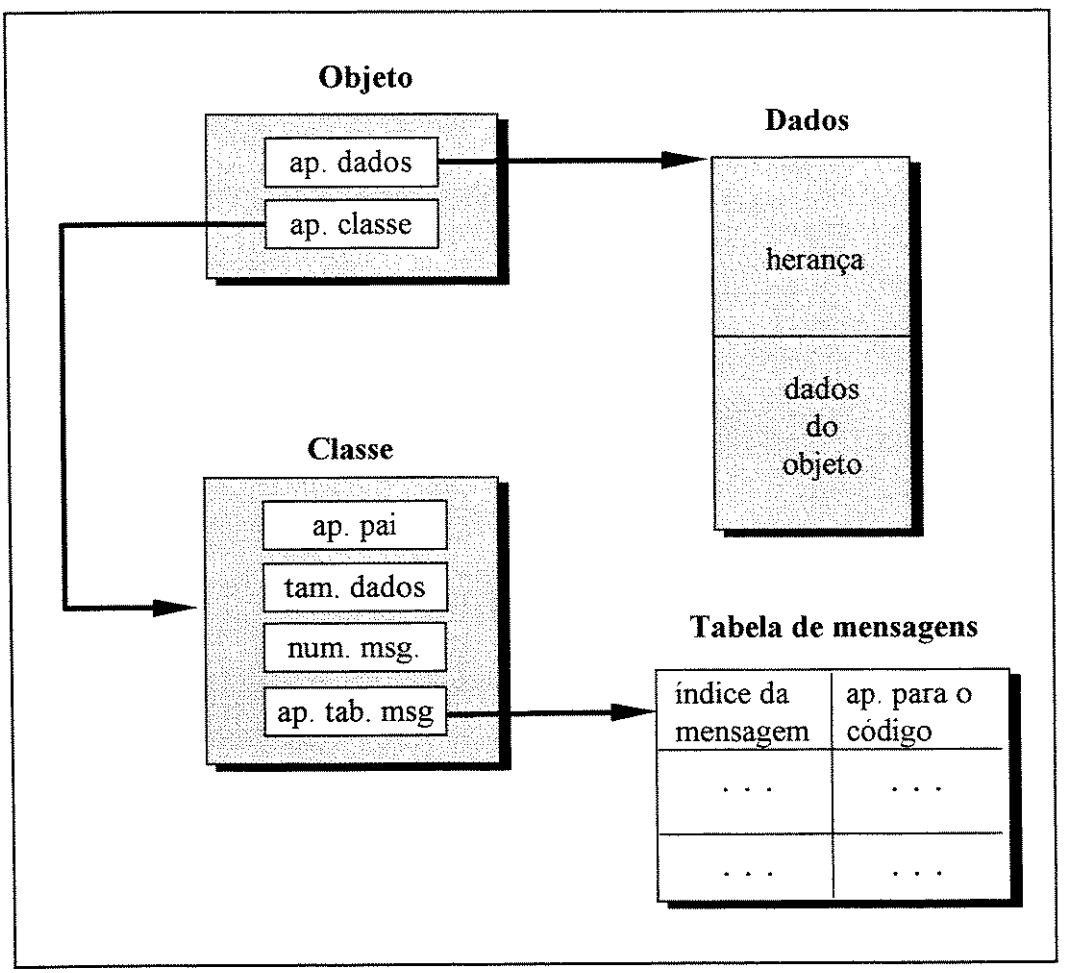

Figura 1.2: O núcleo de implementação de uma linguagem orientada a objetos

Deve-se ainda salientar, que esses dois métodos foram propostos para que o programador de linguagem C pudesse dispor dos mecanismos da programação orientada a objetos na construção de seus programas. Esses métodos permitem, dentro de uma disciplina estabelecida, que o programador consiga criar programas usando os conceitos de encapsulamento de dados e de reutilização de código. Portanto, grande parte dos prejuízos

com memória e processamento, impostos pelos dois métodos, se deve pelas precauções com a componente humana e pela ausência de um ambiente de programação apropriado com o paradigma da programação orientada a objetos, dentro do universo dos microcontroladores.

1.8. Ambientes visuais de desenvolvimento de *software* para microcontroladores

Atualmente, as ferramentas de desenvolvimento de *software*, amplamente difundidas para microcontroladores, são os ambientes integrados de desenvolvimento. Eles agrupam um editor de programa textual, um compilador acoplado a um montador, uma biblioteca de funções, um linker e, em alguns casos, um simulador para ajudar no processo de depuração. Apesar das tentativas mais recentes de melhorar a interface com o programador, promovendo um ambiente visual de desenvolvimento, esses ambientes de programação continuam com as mesmas características iniciais: desenvolvimento textual de programas, sem qualquer abstração de detalhes do *hardware* do microcontrolador.

• ONAGRO

Uma proposta diferente foi apresentada por [SOU95], [SOU96a] e [SOU96b]. Trata-se de um sistema tradutor que reconhece uma linguagem gráfica de descrição de algoritmos e possibilita a geração de código em Assembly para microcontroladores. Este sistema, chamado de ONAGRO, incorpora um editor gráfico para a entrada do programa-fonte, que é constituído por operações simbolizadas através de ícones (veja ilustração na figura 1.3). A representação do fluxo de execução é semelhante ao dos fluxogramas tradicionais e a criação dos elementos de dados é assistida por janelas de diálogos.

Diferentemente dos ambientes textuais, o ONAGRO interage com o usuário logo na entrada das instruções, a fim de diminuir erros posteriores de compilação. Além disso, ele permite uma maior rapidez na entrada do programa, pois ele é orientado a ícone e não a texto, como nas linguagens convencionais.

O ambiente ONAGRO é muito intuitivo e amigável. A documentação dos programas é feita em tempo-real, visto que o próprio programa-fonte se constitui em uma boa ferramenta de inspeção.

Apesar das vantagens oferecidas, o ONAGRO, para aplicações mais complexas, pode tornar os programas ou muito grandes, do ponto de vista do número de ícones, ou muito confusos, do ponto de vista da legibilidade. Isto ocorre porque, por um lado, a maior parte das operações que a linguagem oferece são operações básicas, sendo necessário um número maior de ícones para descrever operações mais complexas, e por outro, devido à estrutura dos fluxogramas, os algoritmos complexos podem ser expressos usando muitos desvios de execução, o que torna o programa de difícil compreensão.

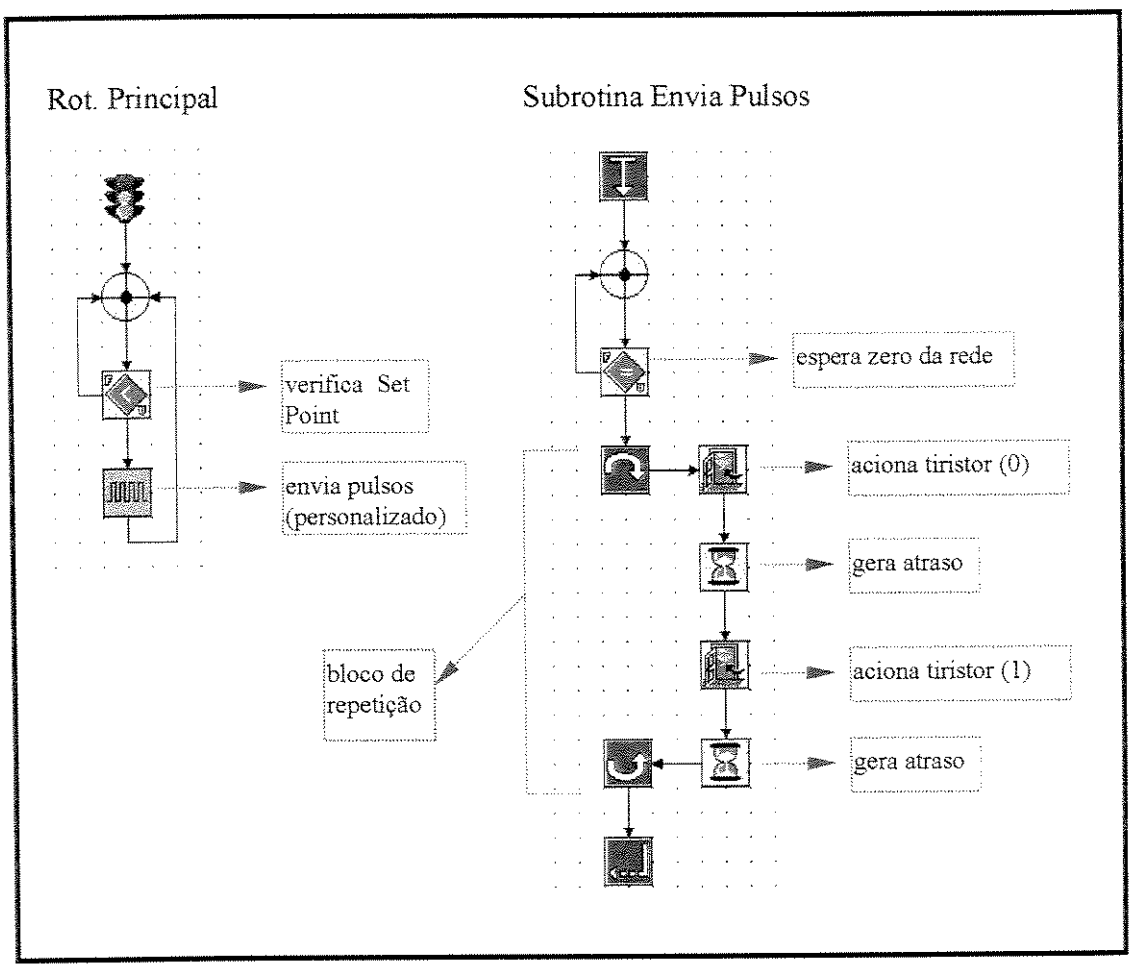

Figura 1.3: Um programa exemplo em ONAGRO

Uma proposta melhorada e incorporando ferramentas para simulação foi proposta em [SIL98] (veja figura 1.4). Nela, os algoritmos são, inicialmente, descritos em um nível mais alto de abstração e, posteriormente, descritos com ícones de mais baixo nível. Assim,

dois níveis de leitura podem ser feitos: um mais do ponto de vista macroscópico, dando ênfase às funcionalidades e outro mais detalhado, dando ênfase aos detalhes de implementação. Com isso, a legibilidade melhorou bastante e, pelo fato de ter incorporado ferramentas de simulação, tornou o processo de desenvolvimento menos artesanal.

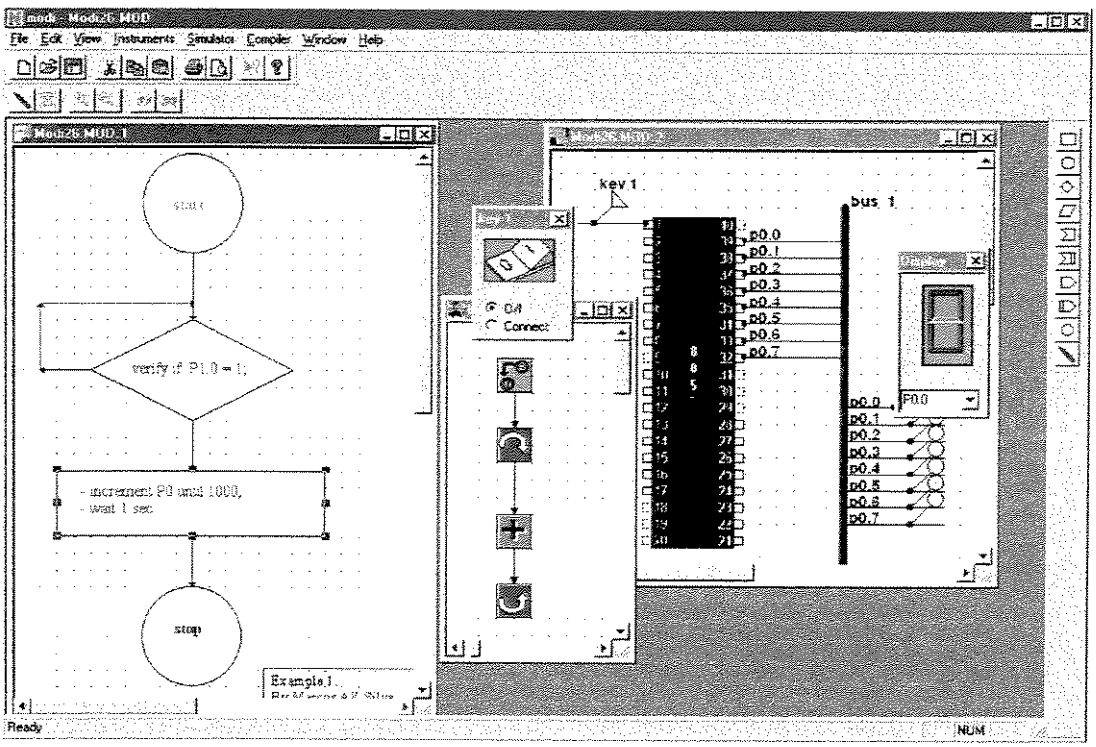

Figura 1.4: O ambiente de simulação MODI

- APBUILDER

Uma outra tentativa de utilizar um ambiente visual para o desenvolvimento de *software* para microcontroladores foi feita, recentemente, pela Intel. A ferramenta, batizada de ***ApBuilder***, foi projetada com a intenção de diminuir a curva de aprendizagem do desenvolvedor de *software* para sistemas dedicados e reduzir o tempo total de desenvolvimento das aplicações.

O *ApBuilder* apresenta os componentes do microcontrolador de forma didática. Para isso, ele faz uso, num primeiro momento, de ícones como o elemento principal de comunicação com o programador, como ilustra a figura 1.5. Depois, através de janelas de

diálogos (veja exemplo na figura 1.6), o *ApBuilder* permite que o programador selecione os parâmetros de configuração dos componentes do microcontrolador e obtenha o respectivo código gerado em *Assembly* ou C.

O *ApBuilder* oferece uma boa variedade de informações *on-line* a respeito do microcontrolador e de seus periféricos. Porém, ele não pode ser visto como um ambiente completo de desenvolvimento, pois ele apenas gera fragmentos de código e os envia ao *clipboard*. É preciso que o programador, por meio de um ambiente textual de desenvolvimento, junte esses fragmentos numa seqüência plausível, e realize as alterações para implementar as interfaces necessárias.

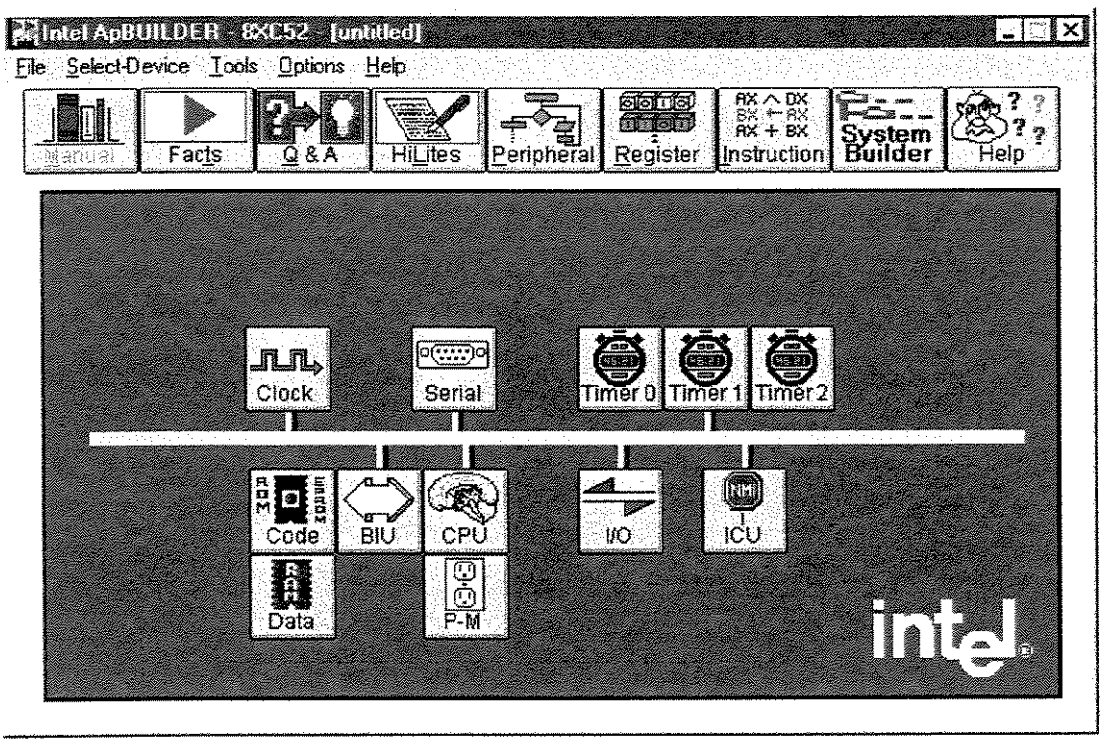

Figura 1.5: A janela principal do Intel *ApBuilder*.

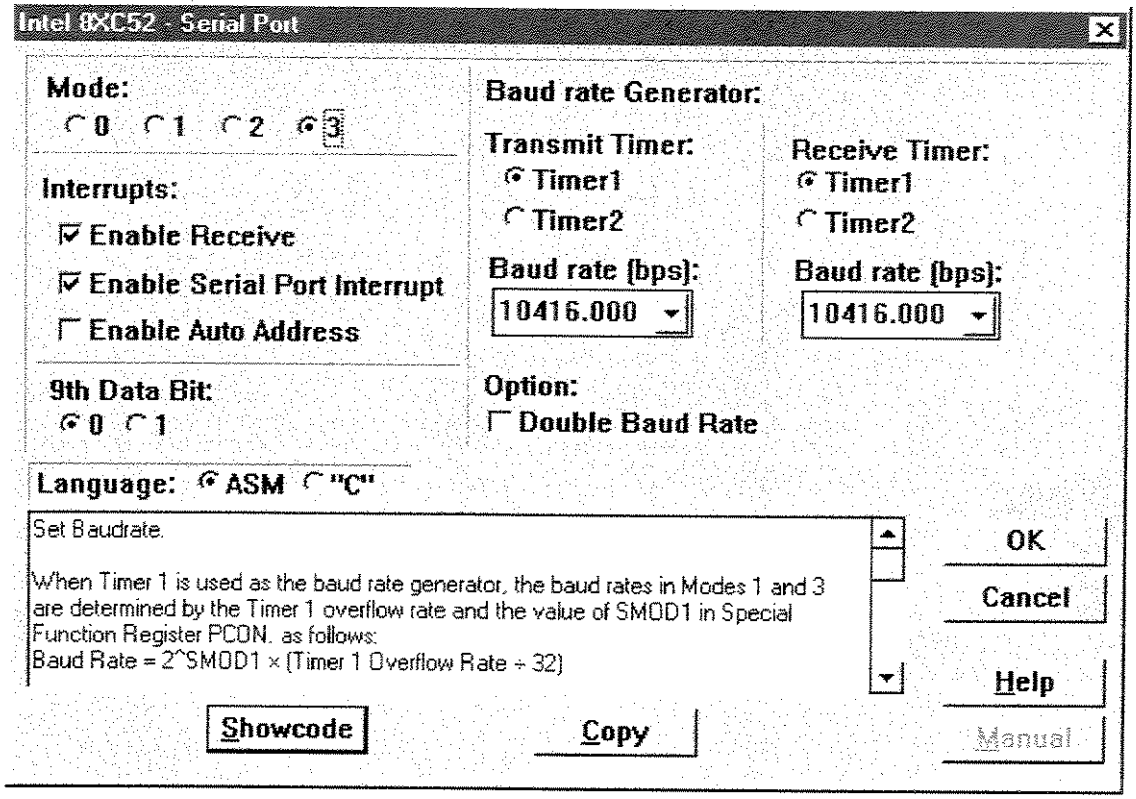

Figura 1.6: Exemplo de janela de diálogo usada no Intel *ApBuilder*.

1.9. Considerações finais

Pelo que já foi exposto, percebe-se que, em se tratando de linguagens de programação para microcontroladores, as linguagens textuais ainda são as mais difundidas.

CAPÍTULO 2

UMA REFLEXÃO SOBRE LINGUAGENS DE PROGRAMAÇÃO

Introdução

Este capítulo dá sustentação teórica para a premissa de que o uso de linguagens visuais pode melhorar a produtividade dos desenvolvedores de *software*. No item 2.1, enfoca-se a evolução dos ambientes de programação textual e os problemas aos quais estão sujeitas as aplicações desenvolvidas nesse tipo de ambiente. No item 2.2, a programação visual é abordada como alternativa para resolver alguns problemas dos ambientes textuais. Relatam-se algumas teorias que apoiam o uso de linguagens visuais e as implicações que podem ocorrer com o seu uso.

2.1. Linguagens textuais de programação

O desenvolvimento do *hardware* dos computadores exerceu forte influência na criação das linguagens de programação. Isto se deve ao fato dessas linguagens terem se desenvolvido em paralelo ao *hardware*. Dessa forma a estrutura das mesmas se moldou às características do *hardware*: E/S baseados em caracter e estruturas seqüenciais.

As linguagens de programação também foram influenciadas pela combinação de formalismos matemáticos com a linguagem natural, que tornou sua sintaxe complexa e inflexível, impondo estruturas restritivas aos algoritmos. Só recentemente é que as linguagens de programação se tornaram mais amigáveis, pois as interfaces com o usuário foram revolucionadas pelo advento dos monitores de alta resolução e de novos dispositivos de entrada, permitindo forte interação com o usuário.

2.1.1. Evolução dos ambientes de programação textual

As linguagens de programação foram criadas com a intenção de representar algoritmos de uma forma precisa e legível. Entretanto, elas foram influenciadas por linguagens usadas para outros propósitos: linguagem natural e formalismos matemáticos (álgebra, cálculo de predicados de 1ª ordem e cálculo lâmbida). Além disso, o

desenvolvimento de linguagens de programação recebeu forte influência do *hardware* e dos sistemas operacionais disponíveis para executar os algoritmos, conforme ilustra a figura 2.1. O *hardware* impôs estruturas restritivas às linguagens devido à sua arquitetura von Neuman, enquanto os incipientes sistemas operacionais permitiam apenas as facilidades de E/S baseada em caracter.

A componente da linguagem natural nas linguagens de programação permite descrever as funções do *hardware*. Por exemplo, variáveis são usadas para simbolizar endereços de memória, palavras-chave como o GOTO e IF-THEN são usadas para descrever desvios do fluxo de execução. Mais recentemente, algumas linguagens descartaram as estruturas da linguagem natural em favor de formalismos matemáticos, como o cálculo lâmbida no LISP e o cálculo de predicados no PROLOG.

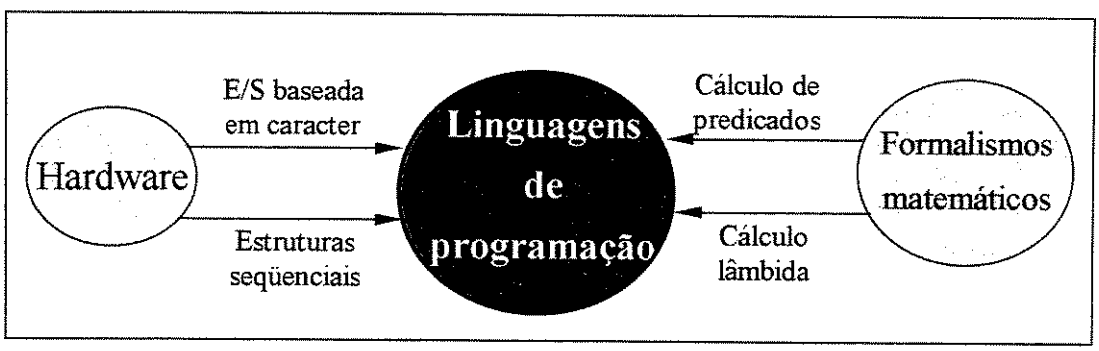

Figura 2.1: Influências sobre as linguagens de programação

Esse histórico desenvolvimento das linguagens de programação teve severas conseqüências indesejáveis. As facilidades dessas linguagens para descrever algoritmos estão mais associadas a como os computadores operam do que com os processos cognitivos e de percepção dos programadores, conforme ilustra a figura 2.2. Como a maioria das expressões textuais é inerentemente unidimensional, os algoritmos são forçados a ser seqüenciais. Esta seqüencialidade desnecessária restringe o pensamento do programador, forçando-o a pensar o programa sempre numa organização linear. Essa mesma linearidade do texto tem um efeito similar na concepção dos dados, já que estruturas multidimensionais (*arrays*) têm que ser representadas em algum modelo linear.

Diferentemente da contrapartida da linguagem natural, a sintaxe das linguagens de programação textual é inflexível, forçando o programador a se deter com pequenos detalhes

sintáticos, ao invés de conceitos importantes do algoritmo. Este problema foi minimizado pelo uso de editores especiais de programa que guiam o programador, fazendo-o se abstrair um pouco mais da sintaxe da linguagem que se estar usando.

A primeira tentativa de descrever mudança do fluxo de controle nas linguagens textuais baseava-se no uso livre dos GOTO's, que foram logo apontados como produtores de programas ilegíveis e propensos a erros. Como resultado dessa experiência negativa, foi introduzida a programação estruturada, enfocando fundamentalmente a necessidade de organizar os programas em módulos, entretanto continuou havendo uma grande cobrança sintática.

Por outro lado, a própria representação de dados nas linguagens textuais pode causar alguns problemas. Valores simples como números e strings são naturalmente representados por texto e portanto, não causam problemas. Todavia, estruturas de dados mais complexas construídas por registros e por apontadores, ou que modelam elementos de características muito peculiares, podem apenas ser representados indiretamente por texto, gerando assim uma semântica artificial de interpretação de dados. Isso conduz, em muitos casos, a ambigüidades e erros de interpretação, dificultando principalmente a manutenção dos programas.

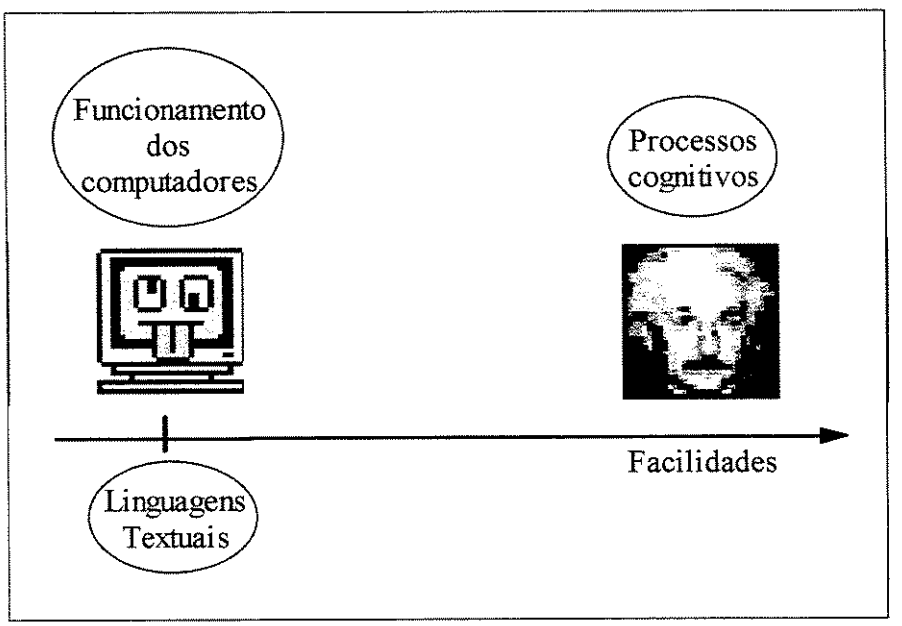

Figura 2.2: Distanciamento das linguagens textuais dos processos cognitivos

2.1.2. Desenvolvimento de *software* em linguagens textuais

Segundo Willian S. Davis [DAV83], em seu livro "Systems Analysis and Design", que se tornou um clássico da literatura na área de projeto estruturado de sistemas, o ciclo de vida de um sistema pode ser sintetizado nas seguintes fases:

❶ Definição do problema e estudo de viabilidade
❷ Análise e projeto do sistema
❸ Implementação e manutenção

Na primeira fase, procura-se definir qual o problema que se quer solucionar e analisar a viabilidade técnica e econômica da solução encontrada, bem como delinear o alcance desta. Após esta avaliação pode-se partir para as fases seguintes ou buscar novos caminhos para resolver o problema.

O passo seguinte consiste em descrever o que deve ser feito para resolver o problema e como este deve ser resolvido. Nesta fase faz-se uso de ferramentas de análise e projeto, como *diagrama de fluxo de dados*, *dicionário de dados*, *fluxogramas* e *diagrama de blocos*, dentre outras.

Por último, tem-se a fase de implementação e manutenção, em que a solução do problema é finalmente escrita em linguagem de computador.

Apesar da abordagem simplista sobre o ciclo de vida de um sistema, que não é o objeto principal deste trabalho, deve-se observar que ao final da segunda fase a solução do problema é descrita, na maioria dos casos, por meio de mecanismos gráficos e ao final da terceira fase, a mesma solução é obtida através de linguagem de formato textual, caso se utilize uma linguagem de programação convencional.

Freqüentemente, em pequenas aplicações ou em aplicações que necessitem de soluções rápidas, é comum se partir diretamente para a fase de implementação, o que torna a documentação do sistema escassa ou quase inexistente, conforme ilustra a figura 2.3. E mesmo nas aplicações que se respeite todo o ciclo de desenvolvimento, as especificações geradas na análise do sistema podem ser mal interpretadas na fase de implementação, pois normalmente estas fases são executadas por pessoas diferentes: o analista e o programador, respectivamente, conforme ilustra a figura 2.4.

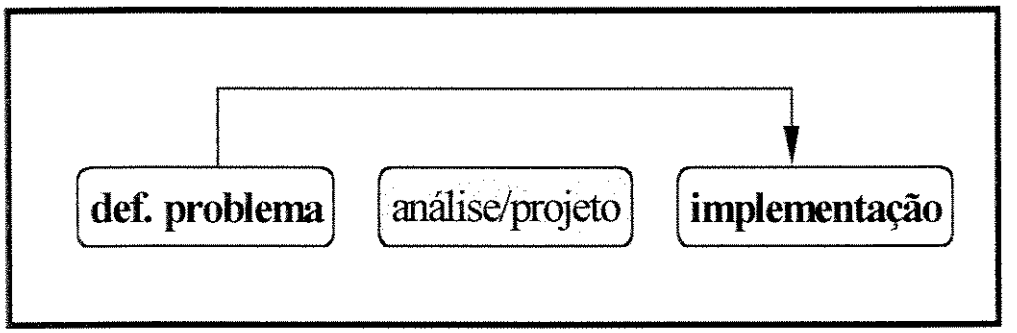

Figura 2.3: Projeto que parte da definição diretamente para a implementação

Isto ocorre porque o processo de codificação de um algoritmo para uma linguagem textual de computadores (como por exemplo BASIC, PASCAL e C++), envolve algumas restrições: em primeiro lugar, deve-se salientar que existe um grande distanciamento entre a sintaxe das linguagens textuais e os mecanismos clássicos de representação de algoritmos, que usualmente usam uma linguagem gráfica, como por exemplo os fluxogramas; e em segundo lugar, se por um lado, as linguagens procedimentais e as orientadas a objetos possuem uma sintaxe que se assemelha à linguagem natural (o inglês) por outro, o grau de detalhamento envolvido em suas operações torna os programas muitas vezes bastante extensos, dificultando a manutenção deste por parte de um programador não envolvido em sua elaboração, conforme ilustra a figura 2.5.

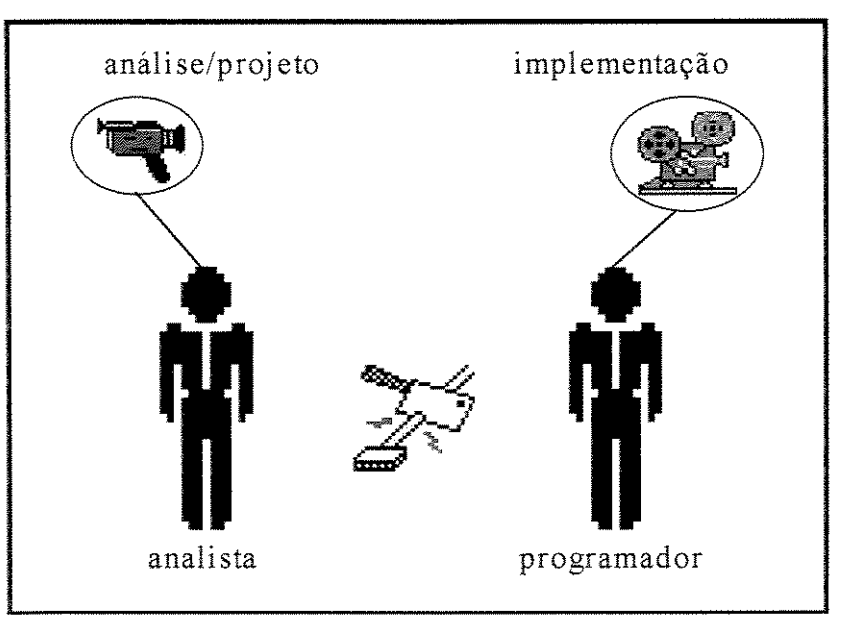

Figura 2.4: Problemas de comunicação entre analistas e programadores

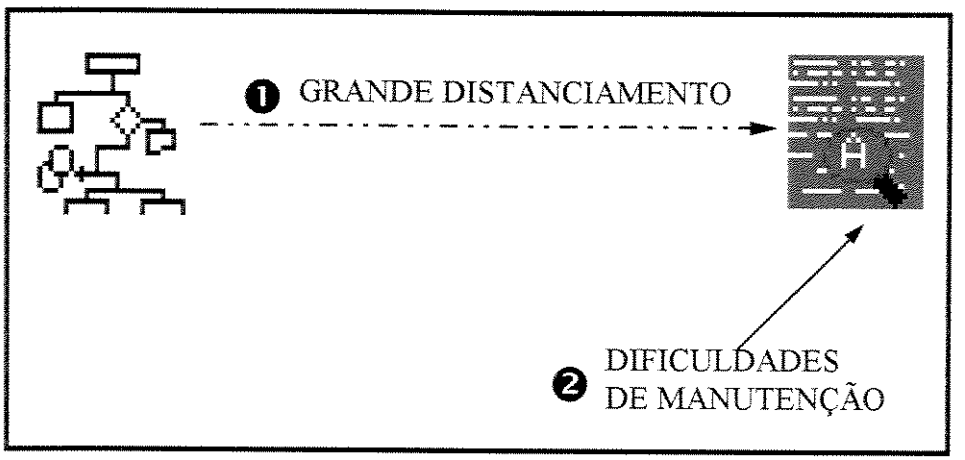

Figura 2.5: Distanciamento entre descrição gráfica e linguagens textuais

2.2. Linguagens visuais de programação

Com o advento da microeletrônica, houve uma profunda alteração nos rumos da utilização dos computadores. Tanto a capacidade de processamento como a quantidade de memória aumentaram em grande escala, ao mesmo tempo que as dimensões físicas e o custo caíram quase que em igual proporção. Isto possibilitou uma maior diversificação de aplicações, permitindo um grande aumento no número de pessoas diretamente envolvidas no desenvolvimento dessas aplicações.

Desta forma, a quantidade de equipamentos disponíveis excedeu ao de especialistas em desenvolvimento de sistemas. Assim um número considerável de aplicações teve que ser implementado por pessoas não especializadas em programação. Scott Brown[LEE95], um estudioso sobre o futuro do *software*, coloca que esse processo representa "a queda da aristocracia do *software*" e o surgimento de uma nova classe de programadores mais envolvidos com a aplicação do que com o *hardware* usado nela. Surge então a necessidade de tornar essa tarefa mais simples e mais próxima das aplicações e desses novos programadores.

Uma alternativa encontrada, em face aos problemas inerentes às linguagens textuais, foi a criação dos compiladores gráficos, que permitem uma geração automática do programa para uma linguagem textual ou para o próprio código de máquina. Com isto se elimina o problema de comunicação entre analistas e programadores e o duro trabalho da fase de codificação. Além disso, o uso de linguagens gráficas torna a documentação sempre

atualizada, pois os compiladores gráficos utilizam-se das próprias ferramentas de documentação (notadamente na forma de diagramas) como fonte de entrada.

Entretanto, o aspecto mais importante das linguagens gráficas pode ser resumido pela frase muito conhecida: ***"uma figura vale por mil palavras"***.

Isto fica evidente se observarmos os *softwares* para microcomputadores, que tendem a usar um ambiente interativo com o usuário, explorando fundamentalmente mecanismos gráficos para melhor implementar a interface homem-máquina.

No caso dos microcomputadores esta tendência tornou-se nítida em meados da década de 80 com a mudança dos sistemas operacionais orientados a comandos de linha para os sistemas operacionais orientados a ícones. Com isso, foi possível uma maior abstração do *hardware* e do próprio *software* (o sistema operacional) por parte do usuário, que para realizar suas tarefas não necessita se deter com tantos detalhes de sintaxe ou conhecer com maior profundidade os mecanismos de acesso aos recursos da máquina.

As linguagens visuais podem simplificar dramaticamente a implementação e manutenção do *software*. Ao expressar programas de forma visual, ao invés de forma textual, pode-se obter ganhos de produtividade semelhantes aos obtidos hoje em dia com as interfaces gráficas com o usuário (GUI) dos sistemas operacionais, como o Macintosh e Microsoft Windows.

2.2.1. Definições preliminares

Informalmente podemos definir uma *linguagem gráfica de programação* ou *linguagem visual de programação* (Visual Programming Language) como sendo uma linguagem que representa visualmente as estruturas usadas na codificação de um programa. Uma linguagem é considerada visual quando ela usa significativamente elementos visuais, tais como diagramas, ícones, gráficos e animação para descrever suas expressões. Uma linguagem cuja utilização destes elementos seja meramente decorativa não é, por assim dizer, uma linguagem visual.[SHU88] [CHA90]

Formalmente, uma linguagem de programação é dita uma linguagem visual de programação quando ela apresenta uma sintaxe visual. Uma linguagem possui uma sintaxe visual quando os elementos terminais de sua gramática são gráficos, como figuras, formas ou animações. A sintaxe visual pode incorporar informações espaciais como *inclusão* ou

conectividade, que realçam o relacionamento entre os elementos da linguagem; e também podem incorporar atributos visuais, como *localização* ou *cor*, evidenciando características particulares dos elementos.[BUR95]

De forma semelhante, podemos definir *ambiente visual de programação* como sendo um ambiente que faz uso de ferramentas visuais no processo de programação. Entenda-se por ambiente de programação, o conjunto de ferramentas e a devida interface com o usuário, usadas para criar, modificar e examinar programas.

Dentro de um contexto mais geral, as linguagens visuais de programação, de acordo com [BUR94], são uma sub-área da "visual computing" , conforme ilustra a figura 2.6. Nesta figura podemos observar, por exemplo, que linguagens visuais de programação e visualização de programas são sub-áreas distintas. Esta última está relacionada apenas com as atividades de testes e manutenção de programas, enquanto que a primeira está relacionada com o processo de programação como um todo.

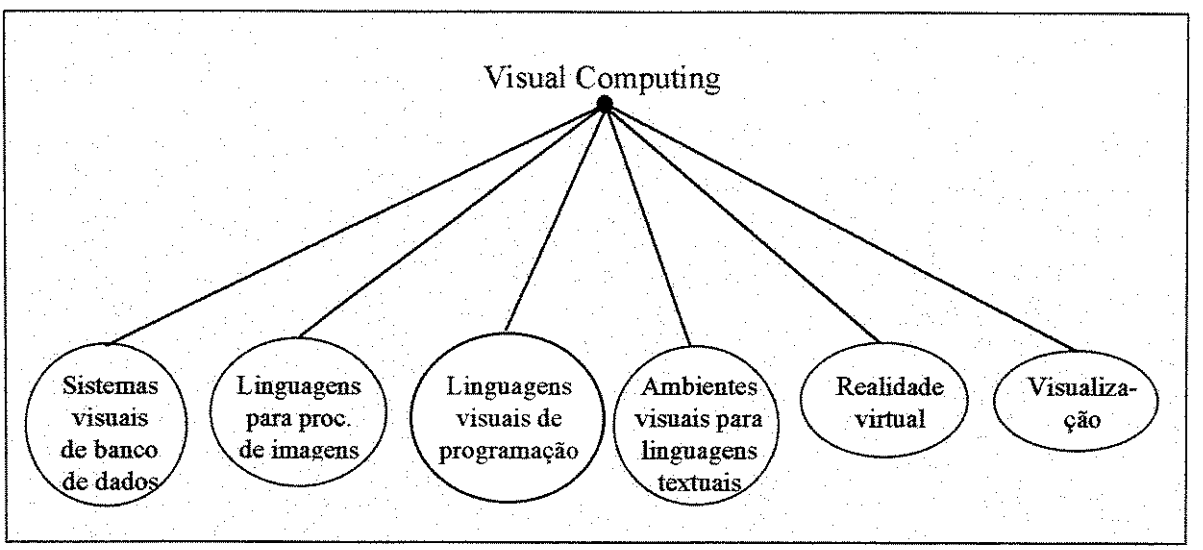

Figura 2.6: O contexto das linguagens visuais de programação

2.2.2. Algumas teorias que apoiam o uso de linguagens visuais

Sabe-se que o ser humano tem maior habilidade em interpretar um algoritmo através da linguagem gráfica ao invés da linguagem textual. É o que ilustra o exemplo apresentado na figura 2.7. A representação gráfica permite observar, de forma direta, os caminhos de

ligação (M) entre as instâncias (I), ao passo que com a representação textual, esses caminhos só são vistos de forma indireta.

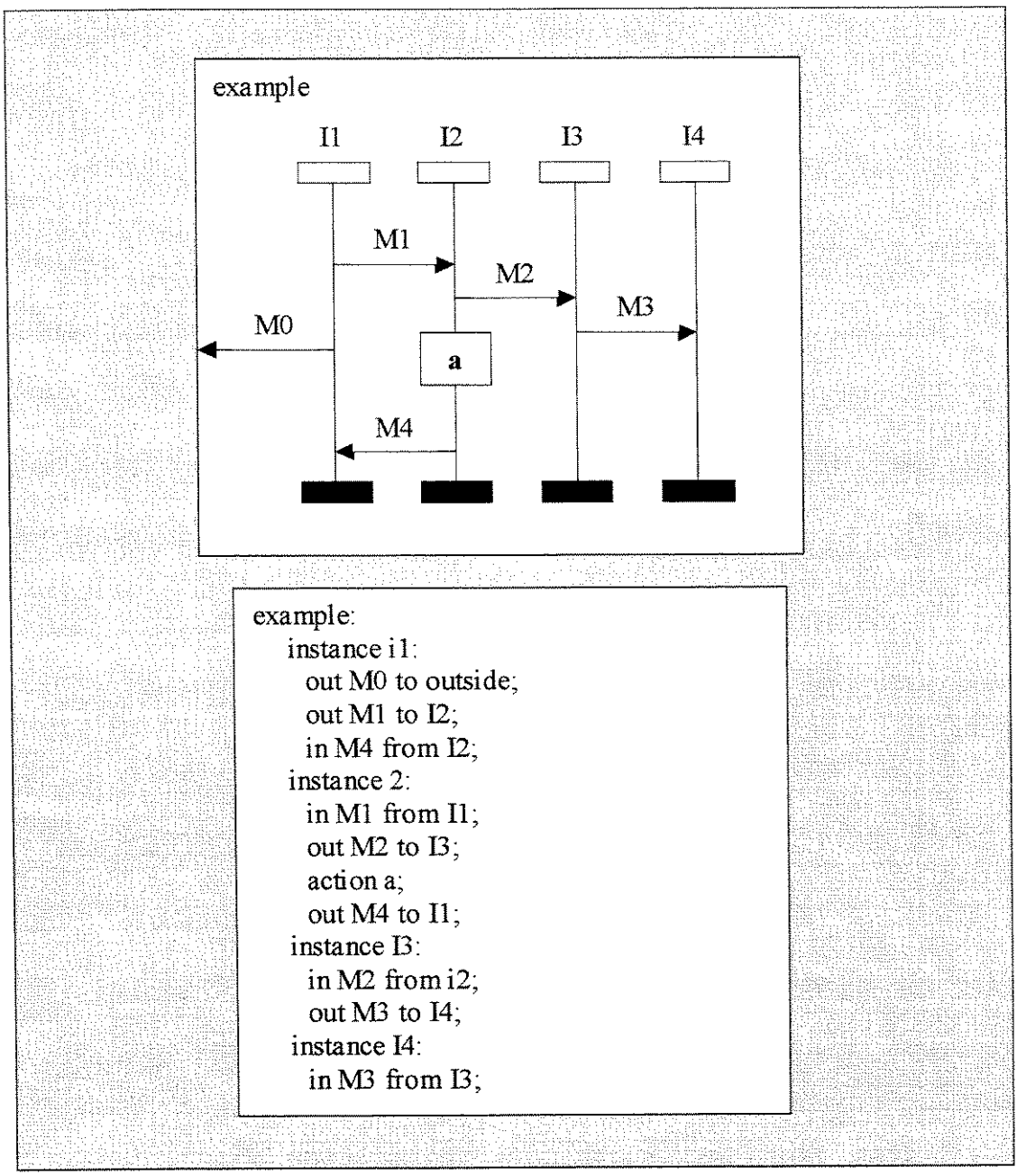

Figura 2.7: A representação gráfica pode ser mais clara que a textual

Pesquisas na área de "neurocognição" (Springer e Deutsch [SPR85]) mostraram que o hemisfério esquerdo do cérebro processa informações seqüencialmente, verbalmente e logicamente. Já o hemisfério direito processa informações simultaneamente, visualmente e

espacialmente. Acredita-se que os atributos visual e espacial formam uma única habilidade cognitiva. Veja ilustração na figura 2.8.

As técnicas textuais ou verbais de descrição de algoritmos, como o pseudocódigo e as linguagens de programação tradicionais, ativam em maior grau os recursos neurocognitivos do hemisfério esquerdo do cérebro. Isto ocorre porque as técnicas textuais de construção de algoritmo contêm mais estímulos seqüenciais, verbais e lógicos, que são os estímulos que mais sensibilizam este hemisfério. [SPR85]

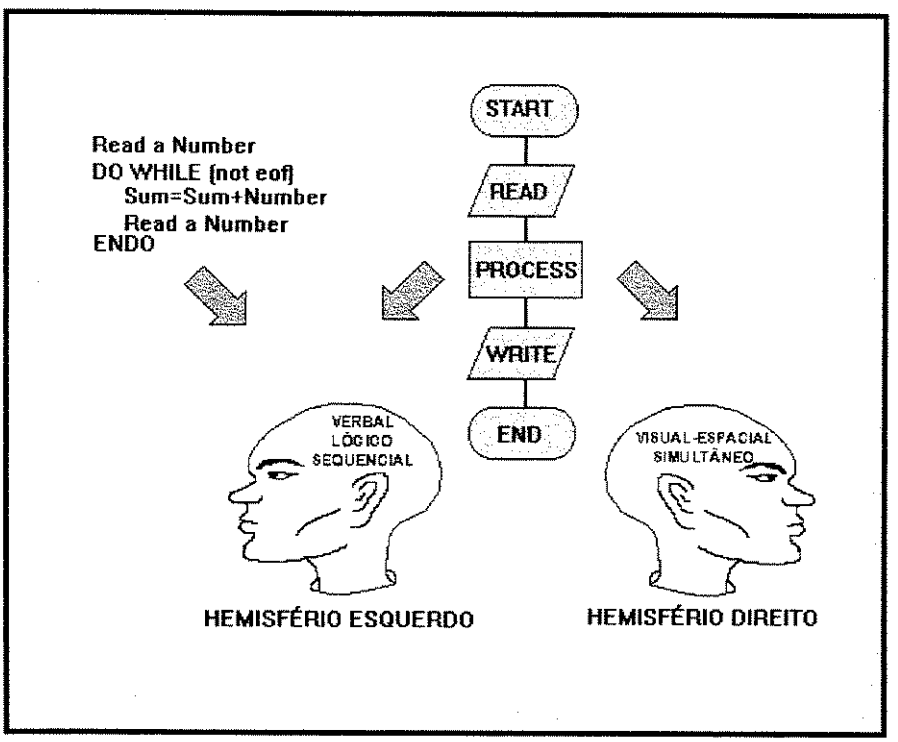

Figura 2.8: O processamento de informações no cérebro

Os mecanismos textuais contêm muito pouca informação espacial, desta forma o hemisfério direito não contribui significativamente com eles. Conseqüentemente, somente metade do cérebro pode estar influenciando no processo de compreensão quando se usa estas técnicas. [SPR5]

Ao contrário das técnicas textuais, os mecanismos gráficos de compreensão de algoritmo, como os fluxogramas estruturados, tendem a estimular todo o cérebro. As técnicas gráficas possuem informações seqüenciais, lógicas e, em menor proporção, informações verbais. Assim possibilita o estímulo do hemisfério esquerdo do cérebro. Além

disso, as técnicas gráficas também estimulam o hemisfério direito, pois contêm muita informação visual-espacial. [SPR85]

Pesquisas realizadas por B. A. Calloni e Donald J. Bagert [CAL94] mostraram que as linguagens gráficas são mais intuitivas e facilitam a aprendizagem e compreensão de algoritmos.

Eles desenvolveram uma linguagem de programação procedimental baseada em ícones para o ensino de programação, chamada BACII e realizaram comparação com a linguagem textual PASCAL, muito usada nas universidades para o ensino de iniciação à programação.

A pesquisa foi motivada pelos resultados obtidos no trabalho, também experimental, do psicólogo David A. Scanlan na área de compreensão de algoritmos, que após testes realizados [SCA88] [SCA89] concluiu que os métodos gráficos possibilitam a abstração necessária dos detalhes de sintaxe e apresentam-se melhores que os métodos textuais nos processos mentais intuitivos para o ensino (e compreensão) de desenvolvimento de algoritmos.

No trabalho de B. A. Calloni e Donald J. Bagert, os alunos submetidos à pesquisa foram alocados, aleatoriamente, para turmas que usaram o BACII e outras que usaram o PASCAL. Após uma análise estatística do desempenho dos alunos, os autores concluíram que a linguagem gráfica foi mais eficaz que a linguagem textual, melhorando a aprendizagem e a compreensão de algoritmos.

Um outro aspecto importante das linguagens visuais decorre da boa adequação desse tipo de linguagem às características psicológicas desejáveis de uma linguagem de programação. Segundo B. Shneiderman [SHN80], uma linguagem de programação deve-se concentrar em preocupações humanas tais como: facilidade de uso, aprendizagem simples e freqüência de erro reduzida. Esses três pontos estão intimamente relacionados com o grau de satisfação do usuário, que é um fator decisivo para o sucesso do uso de uma linguagem. As facilidade de uso e de aprendizagem das linguagens visuais já foram comentadas anteriormente. No que concerne a freqüência de erro reduzida, pode-se observar que as linguagens visuais, na maioria das vezes, possuem uma sintaxe dirigida por um editor gráfico. Desta forma, quase sempre, só são permitidas combinações corretas de declaração.

Isto diminui excessivos testes de sintaxe e, por conseqüência, torna a geração de erro menos freqüente.

2.2.3. O uso de ícones em programação

O uso de ícones sempre esteve associado com computadores e programação. Antes mesmo do computador digital, a programação de máquinas analógicas era feita de forma visual, através de diagramas que especificavam as conexões entre os elementos envolvidos no processamento. Os ícones também exerceram um papel chave na programação dos computadores digitais; por exemplo, os fluxogramas são comumente usados como ferramenta de notação no desenvolvimento de algoritmos; e os diagramas estruturados de dados são usados na definição de estruturas de dados complexas.

A razão primária para o uso de ícones na programação dos computadores é dotar os programas de uma dimensão física, que os aproxime do modelo real. Isto não ocorre com as linguagens textuais. Os elementos de um programa em uma linguagem textual não têm manifestação física, sendo referenciados somente por identificadores textuais. Por exemplo, o nome de uma subrotina usado em uma instrução de chamada não possui, por si mesmo, nenhuma característica que identifique a referência como uma subrotina. Em uma linguagem visual, entretanto, o ícone de um elemento do programa pode comunicar essas características, enquanto o nome é usado apenas para distingui-lo de outros da mesma categoria.

O uso de ícones também pode exercer um papel importante na representação de dados. A maioria das linguagens pode apenas representar dados simples, ao passo que o uso de linguagens visuais permite que dados complexos possam ser representados em vários níveis de abstração, fornecendo uma semântica apropriada para cada aplicação em particular. Por exemplo, uma instância da classe ***livro*** poderia ser representada pelo ícone 🕮, ou uma instância da classe ***avião*** pelo ícone ✈.

2.2.4. Linguagens visuais comerciais

Apesar de ser uma área recente de pesquisa, a programação visual vem se consolidando como uma alternativa atraente para o aumento de produtividade no

desenvolvimento de *software* para aplicações do "mundo real". Estudos realizados no Measurement Technology Center (Raleigh, NC, USA) mostraram que o uso de linguagens visuais, em desenvolvimento de aplicações de aquisição e análise de dados, pode ser de 4 a 10 vezes mais produtivo do que o uso de linguagens textuais [BUR95]. Para essas aplicações que manipulam sinais do mundo físico (instrumentação, controle, monitoramento, etc), podemos destacar duas linguagens comercialmente disponíveis: o Visual Enginnering Environment da Hewlett Packard (HP VEE) e o LabVIEW na National Instruments. Este último se consolidou como o principal *software* do mercado na área de instrumentação.

- LabVIEW (Laboratory Visual Environment Workbench)

O LabVIEW é um ambiente visual para desenvolvimento, depuração e execução de programas. Inicialmente ele foi projetado para trabalhar apenas com placas de aquisição da National Instruments usando a plataforma Macintosh. Entretanto, hoje em dia, seu alcance está bastante ampliado. O LabVIEW é considerado uma linguagem de propósito geral, sendo muito usado em aplicações de aquisição de dados, controle, simulação, processamento de sinais e análise. Há versões para plataformas Mac, Sun e PC.

O LabVIEW não permite programação orientada a objetos, sendo uma linguagem orientada ao fluxo de dados. Ele se baseia no conceito de **instrumento virtual**, que permite, através de *software*, transformar computadores convencionais, portadores de placas de aquisição de dados, em instrumentos de teste e medida. Ele trabalha com duas etapas de definição da aplicação (VI-Virtual Instrument): a primeira cria uma interface com o usuário da aplicação (ver figura 2.9), através de painéis frontais (*front panel*), estabelecendo os controles e indicadores da aplicação (botões, gráficos animados, medidores, caixas de texto, dentre outros); e a segunda descreve o diagrama funcional da aplicação (ver figura 2.10), estabelecendo a lógica de programação, desde o processamento das informações de entrada até a geração dos dados de saída. Este diagrama é composto pelos símbolos que representam os controles e indicadores do painel frontal; por uma série de operações e estruturas pré-definidas na linguagem; e módulos (subrotinas) criados pelo programador.

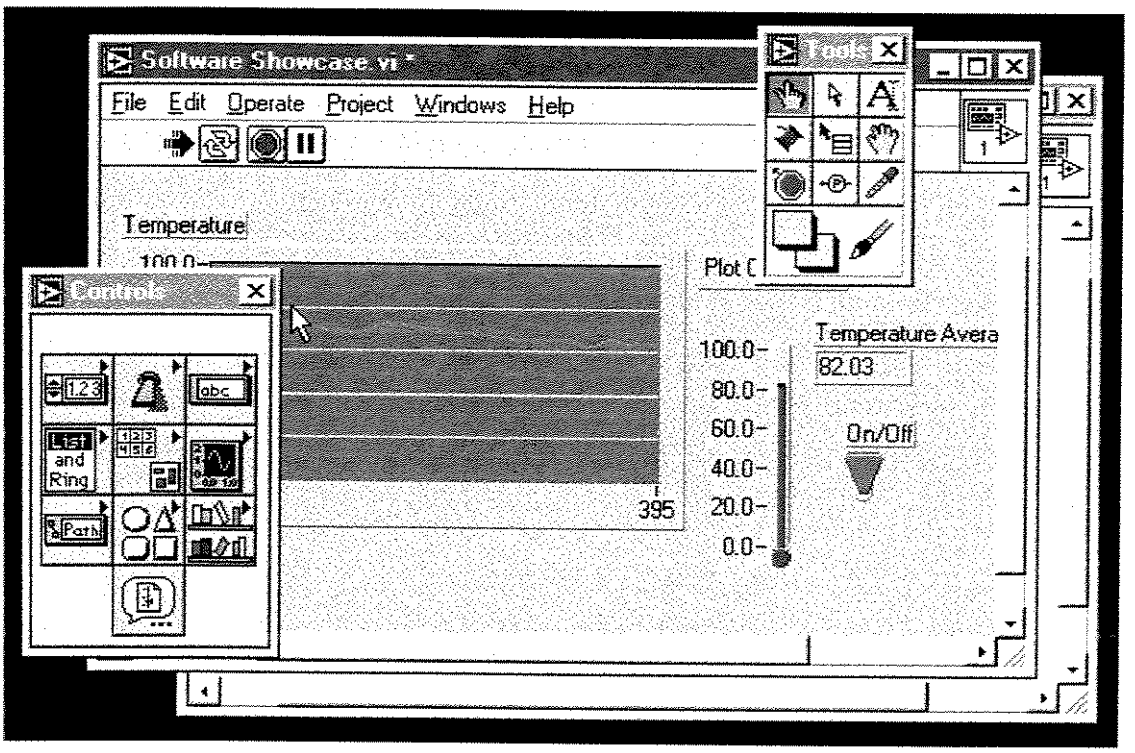

Figura 2.9: Exemplo de um *Front Panel* do LabVIEW

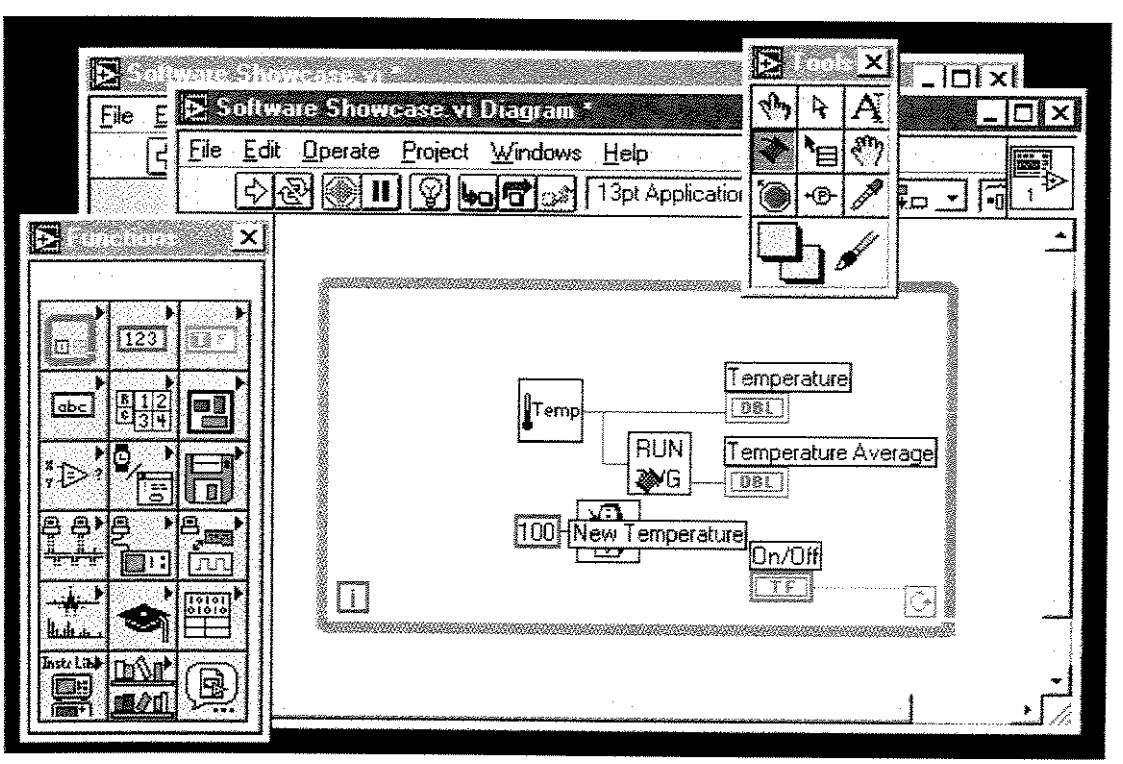

Figura 2.10: Exemplo de uma janela de diagrama no LabVIEW

2.2.5. Limites da programação visual

As linguagens de programação textuais, bem como o *hardware*, não dispõem de muitas facilidades para representar dados e algoritmos, entretanto em muitos casos o uso de textos é superior ao uso de gráficos: rotular, comentar, quantificar, identificar elementos do programa e expressar formulas algébricas [GRE92] [NIC94].

Além disso, dependendo do universo intelectual das pessoas envolvidas no uso de uma linguagem visual, alguns símbolos usados na linguagem podem provocar confusão e conduzir a erros de comunicação, o que torna a linguagem imprecisa.

Em 1915, o psicólogo dinamarquês Edgar Rubin introduziu uma imagem (figura 2.11) que exemplifica a possibilidade de duplicidade de interpretações de um símbolo. No caso ilustrado pela figura 2.11, uma pessoa poderia ver duas faces e outra pessoa poderia ver um cálice.

Essa possibilidade de imprecisão de uma linguagem visual é um grande risco ao seu sucesso, principalmente em se tratando de linguagens de propósito geral, onde o universo intelectual dos programadores pode não ser o mesmo. Já no caso de linguagens visuais com domínio de aplicação específico, esse problema é bastante minimizado, visto que os símbolos usados na linguagem se constituem, preferencialmente, em imagens do universo dos profissionais daquele tipo de aplicação.

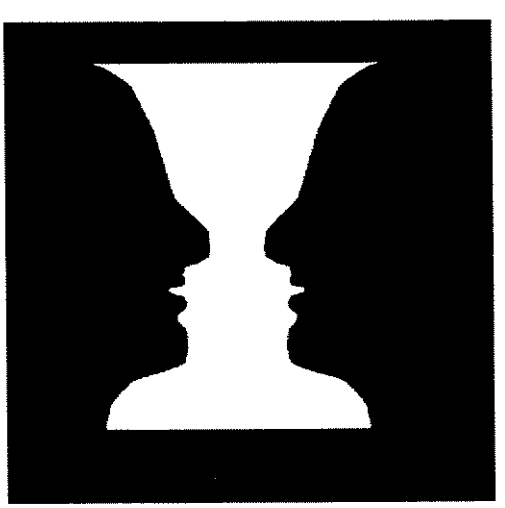

Figura 2.11: Um símbolo sujeito a confusão

Um outro ponto sujeito a controvérsias a respeito das linguagens visuais é a dificuldade de se expressar visualmente algoritmos recursivos. No caso do cálculo de fatorial, a representação gráfica pode fazer uso de uma redução da imagem original do algoritmo (figura 2.12) ou ligar, através de uma linha, o ponto de chamada, dentro da função fatorial, ao ponto de entrada desta mesma função (figura 2.13). No primeiro caso, as reduções consecutivas tornam a representação gráfica imprecisa, uma vez que os detalhes vão sendo perdidos a cada nível. Já no segundo caso, é preciso um esforço do programador para perceber que, visualmente, está ocorrendo uma chamada recursiva. Usualmente, algoritmos recursivos são melhor expressos através de fórmulas algébricas.

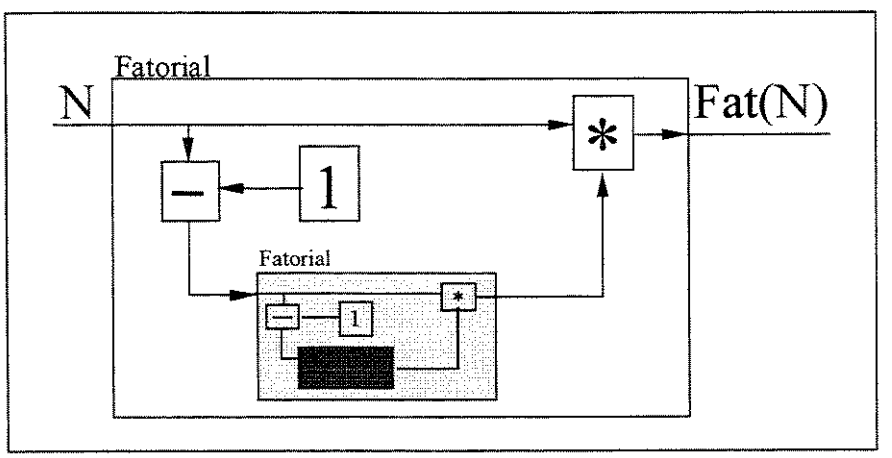

Figura 2.12: Fatorial representado graficamente (caso 1).

A medição de produtividade em linguagens visuais é um outro ponto de limitação desse tipo de linguagem. Se for utilizada a contagem do número de linhas de código (LOC) pode haver inconsistências nessa medição, haja visto que um elemento visual pode corresponder a várias linhas de código. Neste caso, uma abordagem mais qualitativa pode levar a resultados mais consistentes, no entanto não há nenhum método amplamente aceito para este propósito.

Finalmente, é importante observar a deficiência das linguagens visuais, no tocante a quantificação dos dados. A representação visual de dados é muito útil nos processos cognitivos de comparação. Entretanto, dependendo do grau de precisão utilizado, podem ser geradas confusões nesses processos, principalmente quando forem comparados dados com valores muito próximos.

Pode-se perceber, pela ilustração de dois exemplos apresentados na figura 2.14, que na primeira situação, a representação visual é suficiente para concluir que x é maior que y (4 > 2). Entretanto, na segunda situação, a representação visual pode conduzir a uma conclusão errada, a de que x seria igual a y (4 = 3,9999).

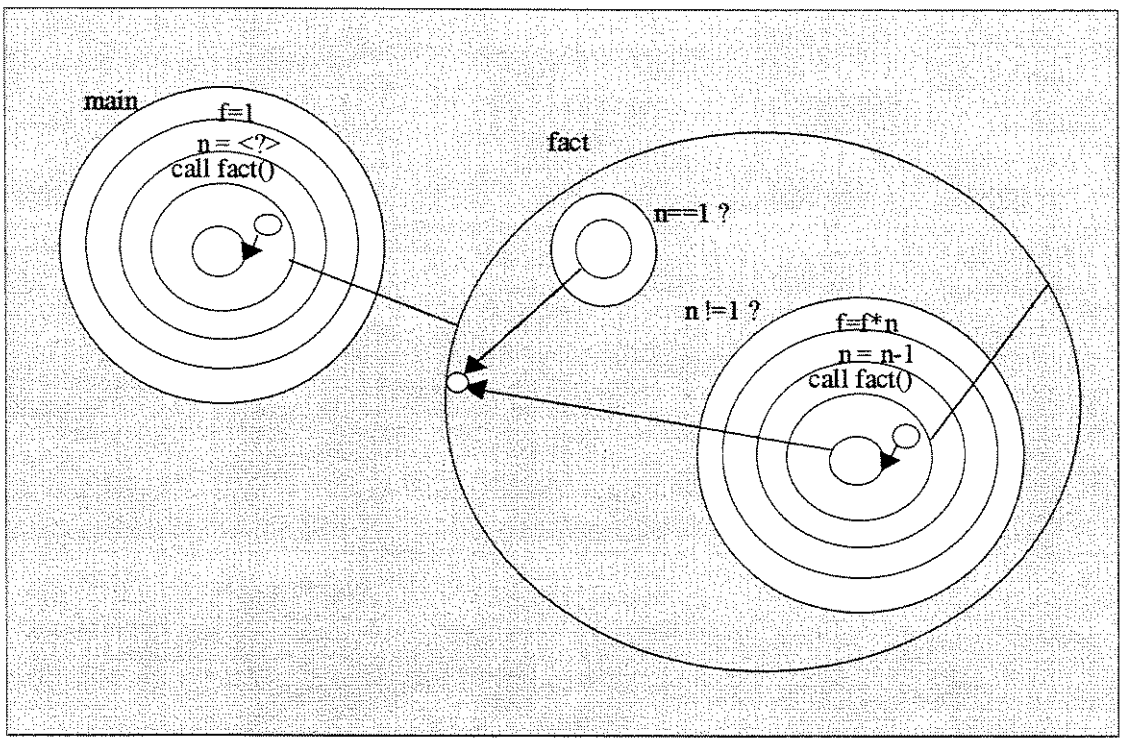

Figura 2.13: Fatorial representado graficamente (caso 2).

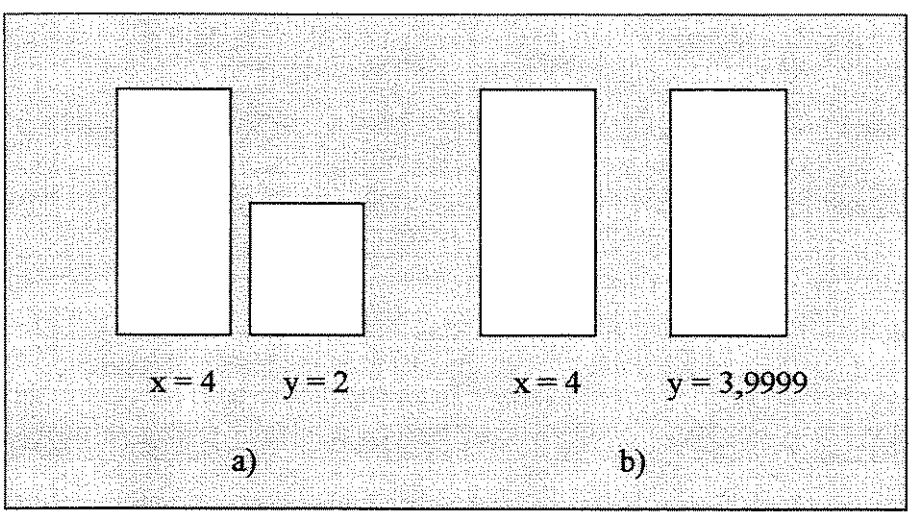

Figura 2.14: A falta de precisão na representação visual dos dados (caso 1)

Neste mesmo sentido, a própria forma geométrica usada para representar os dados pode provocar falsas deduções. A figura 2.15, por exemplo, mostra o caso de duas áreas iguais, mas que, vistas rapidamente, nos induz a pensar que a primeira, da esquerda para a direita, é maior que a seguinte.

Concluindo, deve-se salientar que é uma abordagem pobre, pensar que o visual é sinônimo de "melhor" ou "mais intuitivo". Uma figura pode ser expressiva, mas pode não ser precisa. Assim a escolha dos símbolos usados para representar as operações, como também o modelo de visualização dos dados escolhido é muito importante e deve procurar minimizar as limitações da linguagem visual.

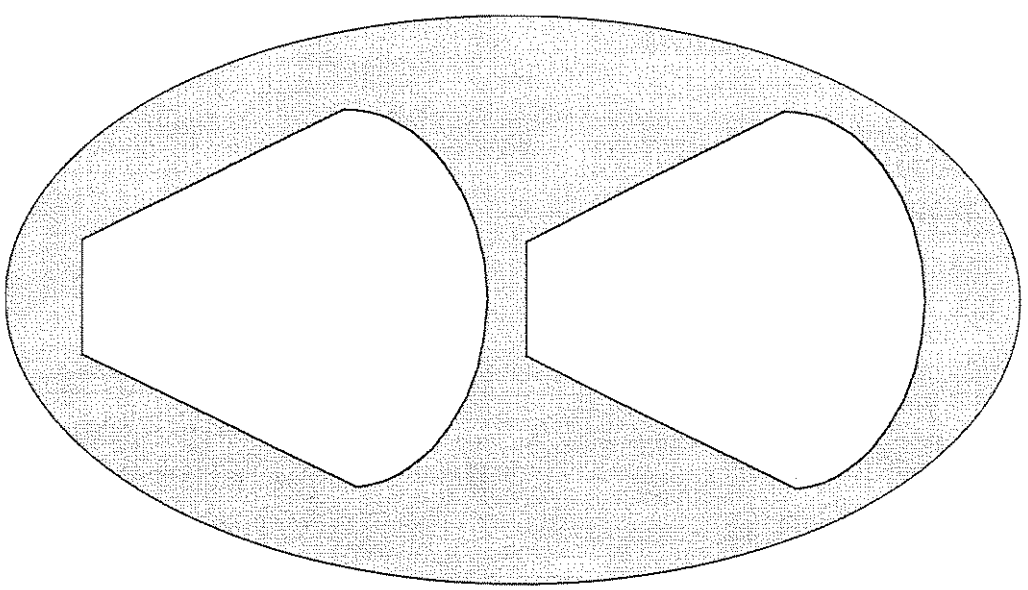

Figura 2.15: A falta de precisão na representação visual dos dados (caso 2)

CAPÍTULO 3

ASPECTOS DA EVOLUÇÃO DO *SOFTWARE*

Introdução

Este capítulo fornece os elementos teóricos para a segunda premissa, o uso da programação orientada a objetos pode, também, melhorar a produtividade dos desenvolvedores de *software*. Neste capítulo é feito um apanhado histórico da evolução do *software*, apontando os aspectos que fizeram uma convergência para a engenharia de *software* e para a programação orientada a objetos.

3.1. *Software* como arte

No início, a programação dos computadores era exercida por verdadeiros gênios artesões. Um programa típico se constituía de uma longa lista de instruções escritas linha por linha em uma linguagem de programação arcaica. Cada parte do *software* era projetada e implementada segundo as necessidades específicas dos usuários. Os desenvolvedores de *software* não dispunham de ferramentas de apoio, nem de metodologias de programação.

Nessa fase do desenvolvimento do *software* dos computadores, as linguagens de programação eram de baixo nível, requerendo muitas instruções para que o computador executasse operações simples. Como o *hardware* era caro, havia sentido em não se usar linguagens de alto nível, pois elas por um lado aceleravam o desenvolvimento do *software*, mas por outro, criavam maiores exigências de memória e de processamento.

Na primeira era do *software* os programas tinham muitas características de arte: eram complexos e, freqüentemente, grandes ou obscuros. Evidentemente, a clareza e a funcionalidade eram deixadas em segundo plano, em detrimento de um compromisso maior com o melhor uso possível do *hardware*.

Além disso, o tamanho desses programas implicava em um longo tempo de desenvolvimento, chegando muitas vezes ao ponto de nunca serem concluídos. Só os verdadeiros "gênios e artistas" conseguiam concretizar suas idéias, que só eles podiam entender. Mesmo assim, a manutenção desses programas pelos seus criadores representava

um difícil desafio, e quando outras pessoas eram designadas para esta tarefa, esse desafio era intransponível.

As incipientes ferramentas de apoio ao desenvolvimento de *software* eram compostas por diferentes linguagens de programação com diferentes "dialetos" e compiladores não padronizados. A portabilidade era impossível e a correção de erros era uma verdadeira neurose.

A despeito das irrefutáveis habilidades dos primeiros programadores, chegou-se a conclusão que *software* não era arte. O cliente patrocinador do *software* precisa de programas bons de se usar, consistentes e funcionais que ajudem a resolver seus problemas. O desenvolvimento desses programas deve ser, na medida do possível, rápido e a custo mínimo. Eles devem ser flexíveis ao ponto de permitir mudanças de forma rápida e de baixo custo. Naturalmente, os clientes desejam programas esteticamente organizados, mas não desejam aprender um novo tipo de interface a cada nova aplicação.

Segundo Don Tapscott & Art Caston [TAP93], o desenvolvimento de *software* como arte implica em sete problemas severos:

1. *Limitada reusabilidade*: é tão difícil se reusar uma parte de um programa da primeira era do *software*, como também o é reusar uma seção de uma obra de algum pintor. Disso decorre o alto custo e o longo tempo de desenvolvimento do *software*.
2. *Manutenção pesada*: o tempo e o dinheiro gastos na manutenção dos sistemas tornavam-os inviáveis.
3. *Inexistência de integração e/ou comunicação*: "sinfonias são trabalhos artísticos isolados". A ligação de dois ou mais programas era uma atividade inviável e inconcebível devido a complexidade envolvida.
4. *Interface com o usuário não padronizada*: "cada peça de arte tem uma estética diferente". O usuário tinha que se adaptar a cada interface de um novo programa. Esta interface herdava características peculiares ao desenvolvedor do programa.
5. *Inexistência de portabilidade*: os programas eram executados em uma plataforma exclusiva, não sendo possível a portabilidade para diferentes plataformas.
6. *Apenas os programadores podiam criar programas*: devido ao elevado nível de especialização, a programação era exercida apenas por profissionais altamente qualificados.

7. *Desenvolvimento orientado ao processo*: as informações eram modeladas tomando-se por base as decisões envolvidas no processo, o que tornava elevado o custo de manutenção.

3.2. *Software* manufaturado

A necessidade de maior produtividade e qualidade no desenvolvimento de *software* torna obrigatória a adoção e uso de modelos padronizados de desenvolvimento. Tal qual uma linha de montagem, o desenvolvimento de *software* precisa de etapas bem definidas, complementares, interligadas e executadas numa seqüência plausível. Evidentemente *software* não é uma geladeira ou um vídeo-cassete. O excesso de disciplina e de métodos usados em seu desenvolvimento pode tirar a criatividade do programador, podendo gerar um *software* de baixa qualidade.

3.2.1. Programação estruturada

A programação estruturada e mais tarde a análise estruturada, se constituíram um passo efetivo na busca de um desenvolvimento de *software* mais sistematizado e produtivo. Ela foi introduzida pela realização de subprogramas como um mecanismo de abstração para elaborar grandes e complexos sistemas. O mais antigo e familiar dos processos da programação estruturada está na decomposição funcional. Como o nome diz, decomposição funcional visualiza um sistema como um conjunto de áreas funcionais que pode ser dividido em processos. Estes processos são depois decompostos em passos menos abstratos, mas compreensíveis pelos computadores. A decomposição de uma função produz processos e subprocessos que basicamente resultam em partes procedimentais.

A decomposição funcional conquistou grande popularidade desde que foi lançada no final dos anos 70, principalmente como ferramenta de análise e projeto e não apenas de implementação. Métodos populares e notações para decomposição funcional são identificados pelos nomes de seus autores: Yourdon, Constatine, DeMarco e Gane. Para sistemas em tempo real, os métodos de decomposição funcional foram ampliados por Hatley-Pirbhai e Ward-Mellor.

Por causa da existência de padrões e softwares para suportar o desenho dos projetos e sua documentação, a técnica de decomposição funcional tem sido muito adotada. Entretanto, a decomposição funcional tem desvantagens. A técnica de projeto estrutural é direta, mas força os programadores a se concentrarem nas operações, com pouca atenção à estrutura dos dados. Os projetos quase sempre resultam em muito código e poucos dados, porque a organização dos dados é derivada dos processos e das suas necessidades de interação. Além disso, a técnica de projeto não é distributiva; diferentes analistas trabalhando no mesmo projeto podem chegar a resultados diferentes.

Um segundo tipo de Análise Estruturada é referido como resposta a eventos. Esta forma foi introduzida em 1984 por McMenamin e Palmer e mais tarde adaptada para sistemas em tempo real por Ward e Mellor. Diferentemente da decomposição funcional, a análise de resposta a eventos focaliza-se em eventos externos para derivar os processos do sistema. O sistema é visualizado como uma caixa-preta que responde a eventos ocorridos fora dele. Cada evento resulta na definição de um processo do sistema. Como na decomposição funcional, a última etapa do projeto é a ligação dos processos com os dados. Os modelos também são de fácil compreensão, porém, geralmente resultam em código excessivo.

Apesar dos enormes avanços estabelecidos pela programação estruturada, muitos dos problemas da primeira era permaneceram vivos, principalmente no desenvolvimento de grandes sistemas.

3.3. *Software* como engenharia

Em meados da década de 80, muitas companhias de *software* começaram a estabelecer padrões de desenvolvimento de *software*, que incorporassem princípios de engenharia usados em outras indústrias, de forma que tornasse viável o desenvolvimento de *software* e permitisse o suprimento da demanda por novos softwares.

Estabelecer princípios de engenharia significa construir melhores ferramentas para automação e refinamento de processos de desenvolvimento, visando a criação de blocos-prontos que podem ser usados em futuros programas. Além disso, a engenharia de *software* visa um gerenciamento mais eficiente e uma regulamentação da disciplina no desenvolvimento do *software*. [TAP93]

Segundo muitos autores, a engenharia de *software* é a chave para migrar do "*software* como arte" para um efetivo desenvolvimento manufaturado de *software*.

Dentro desse conceito de engenharia de *software*, novas metodologias foram propostas, mas a que conquistou maior número de seguidores foi, sem dúvida, a metodologia orientada a objetos.

Apesar de ter permanecido por muito tempo como uma metodologia usada por acadêmicos, hoje a programação orientada a objetos - POO é uma realidade. Segundo um estudo realizado por Edward Yourdon [YOU97], o uso da tecnologia orientada a objetos está em franco crescimento. Em 1991, apenas 3 a 4 por cento dos projetos de desenvolvimento de *software* em todo o mundo estavam usando a tecnologia orientada a objetos; mais tarde, em 1993 seu uso havia sido triplicado; e em 1996 seu uso chegou a, aproximadamente, 60%. As estimativas mostram que até o início do próximo século, o uso da programação orientada a objetos deve chegar a 80% (ver gráfico a seguir).

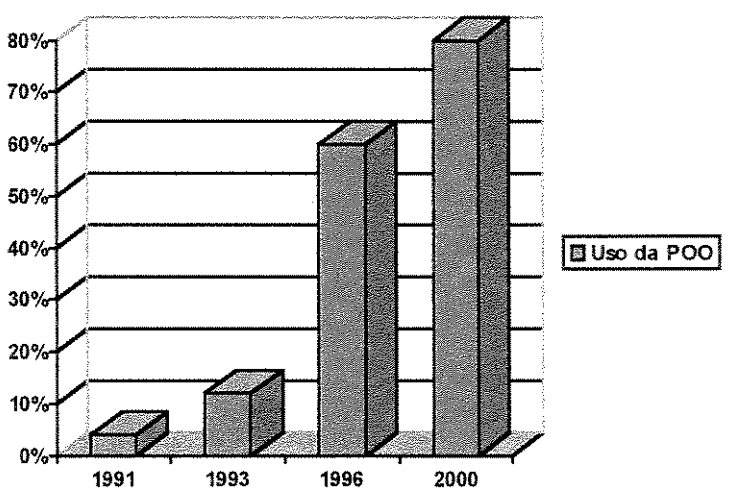

Figura 3.1: Evolução do uso da POO

Entretanto, um levantamento feito por Chris Pickering [PIC96], mostrou que a taxa de crescimento dos projetos bem sucedidos, desenvolvidos com a tecnologia orientada a objetos, não acompanha sua popularização. Em 1991, 91,7% dos projetos desenvolvidos usando a programação orientada a objetos foram bem sucedidos, ao passo que em 1996 esse percentual caiu para 74,0%. Ou seja, o uso da programação orientada a objetos tem

crescido a taxa elevadas, porém os riscos do uso desse paradigma são maiores que, por exemplo, da programação estruturada, que nos mesmos anos obteve taxas de 90,2% e 87,0%, respectivamente (ver gráfico a seguir).

As razões apresentadas para esse aparente sinal negativo da programação orientada a objetos pode ser sintetizada pela falta de treinamento adequado do pessoal envolvido e pelo mau uso de ferramentas de apoio. Evidentemente, o próprio crescimento acelerado da programação orientada a objetos evidencia um certo modismo, ao qual muitos desenvolvedores foram atraídos sem um preparo coerente com suas necessidades.

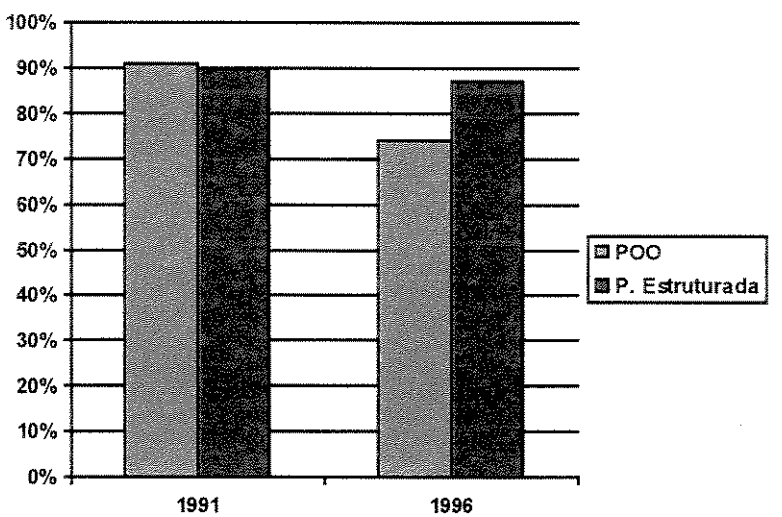

Figura 3.2: Evolução de projetos bem sucedidos

3.3.1. Programação orientada a objetos

Em contraste com a programação estruturada, a programação orientada a objetos trata dados e operações em um único módulo. De fato esse tipo de programação representa informações em unidades chamadas *objetos*, os quais se consistem de dados e de um conjunto de operações para manipular esses dados.

O principal encanto da programação orientada a objetos é que a informação em cada objeto pode ser reutilizada repetidamente em várias aplicações distintas. Classes de objetos são organizadas em uma hierarquia seguindo os princípios de hereditariedade, promovendo assim, uma efetiva reutilização de código.

Dentre as principais vantagens da programação orientada a objetos, em relação às metodologias tradicionais, pode-se destacar:

- ☑ acentuado aumento de produtividade, pois o projeto orientado a objetos tende a aproximar a fase de desenvolvimento à fase de implementação, através de um modelamento mais próximo da realidade da aplicação.

- ☑ o projeto orientado a objetos permite uma maior reutilização do código (operações e dados) já existente, incentivando a modularização conjunta de operações e dados.

3.3.1.1. Conceitos básicos da programação orientada a objetos

Simplificadamente, pode-se dizer que a programação orientada a objetos é uma ferramenta de construção de *software* básico, onde a *reutilização* é o ponto central do desenvolvimento de *software*, pois elaboram-se rotinas reutilizáveis, ao invés de elaborar completamente um programa a partir de definições do mesmo [COX91].

A idéia é colocar a fase de desenvolvimento mais próxima da fase de implementação, aumentando a produtividade de analistas e programadores através de uma maior expansibilidade e reutilização de código; e controlando a complexidade e o custo de manutenção do *software* desenvolvido [RIC91]. Desta forma, a modificação de um programa orientado a objetos não afeta sua estrutura. O objetivo é que cada novo módulo (objeto) introduzido no programa não afete os outros módulos, mas sim reutilize as operações e dados já definidos, conforme mostra a figura 3.3 [WIN91].

Os requisitos básicos para o desenvolvimento de um *software* orientado a objetos são: modularização, abstração e ocultação. Estes requisitos são a base da metodologia orientada a objetos [BOO86].

❶ Modularização

Sabe-se que os métodos convencionais estimulam apenas a modularização do processamento, entretanto o projeto orientado a objetos busca conectar dados e operações,

modularizando tanto o processamento como também os próprios dados.

R. S. Pressman [PRE90] sugere que o *software* deva ser dividido em elementos com nomes e objetivos separados (os módulos) que devem se unir para satisfazer as definições do problema. Com isso poder-se-ia manejar intelectualmente o programa.

Ele também faz uma observação importante quanto à complexidade e ao esforço no desenvolvimento de *software*, conforme mostram as inequações:

a) $C(p_1) > C(p_2) \Rightarrow E(p_1) > E(p_2)$

b) $C(p_1 + p_2) > C(p_1) + C(p_2) \Rightarrow E(p_1 + p_2) > E(p_1) + E(p_2)$

$C \Rightarrow$ Complexidade $E \Rightarrow$ Esforço p_1 e $p_2 \Rightarrow$ Problemas

Pode-se dizer que a modularização procura diminuir os efeitos das alterações em um sistema, desta forma deve-se refinar sucessivamente os módulos, até o ponto em que o esforço para interfaceá-los não comprometa o desenvolvimento do *software*.

❷ Abstração

No modelamento do sistema orientado a objetos tanto os dados como os algoritmos devem ser abstraídos, ressaltando-se a composição e o comportamento dos dados e dos módulos. Deve-se deixar os detalhes de implementação para outras fases, de tal forma que os níveis superiores do sistema estabeleçam uma solução ampla e os níveis inferiores uma solução mais detalhada.

❸ Ocultação

Na programação orientada a objetos deve-se ocultar ou encapsular dados e operações para impedir que as informações contidas em um módulo não sejam acessíveis a outros módulos que não necessitem daquela informação. Assim, torna-se mais fácil a depuração e a manutenção dos módulos, pois dificulta a propagação de erros.

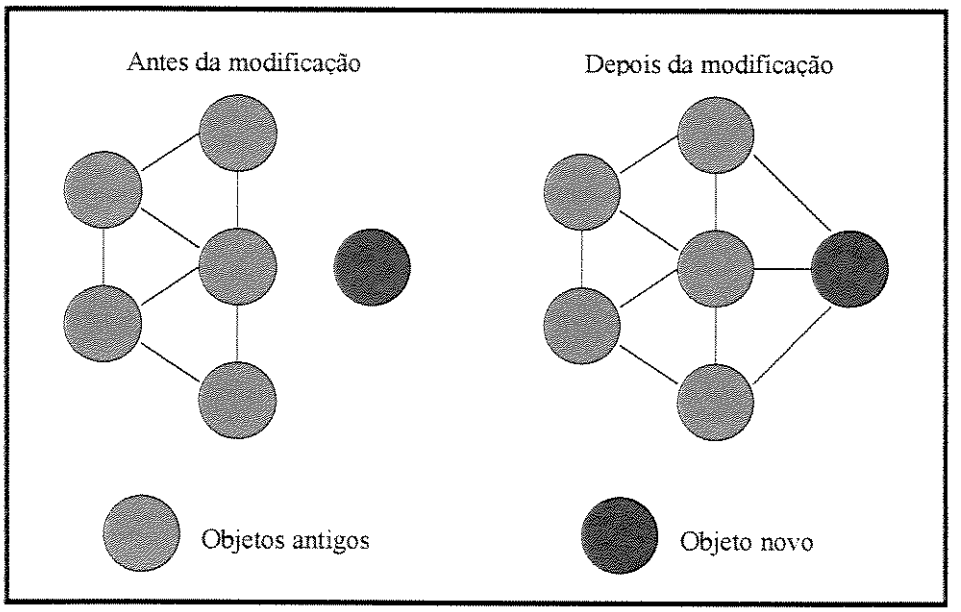

Figura 3.3: A modificação de um programa OO não afeta a estrutura do programa.

3.3.1.2. Estruturas básicas da programação orientada a objetos

Outro aspecto importante da programação orientada a objetos se refere às suas estruturas básicas: os objetos, as mensagens, as classes e a hereditariedade.

Um *objeto* pode ser definido como um elemento do mundo real modelado como um componente de *software*. Todo programa orientado a objetos consiste basicamente em objetos, estes por sua vez são formados por membros de dados e por métodos que manipulam esses dados. Desta forma, pode-se caracterizar os objetos pelos seus atributos particulares (dados) e pelo seu comportamento (métodos).

Outra estrutura importante na programação orientada a objetos é a geração de *mensagens* entre objetos. Pode-se dizer que uma mensagem é uma solicitação para que um objeto assuma um determinado comportamento, ou seja, que um método do objeto que recebeu a mensagem seja executado com base na passagem de um parâmetro ou em um dado do próprio objeto. A figura 3.4 mostra a anatomia de um objeto e ressalta os mecanismos de mensagens como única forma de acesso aos dados do objeto.

Quando um conjunto de objetos apresenta as mesmas características, diz-se que estes objetos pertencem a uma mesma *classe*. Portanto, uma classe representa um modelo pelo qual podem ser criados os objetos. É preciso ressaltar que as classes são entidades

estáticas, só podem ser modificadas em tempo de compilação, ao passo que os objetos são entidades dinâmicas presentes na memória, podendo ser alteradas em tempo de execução.

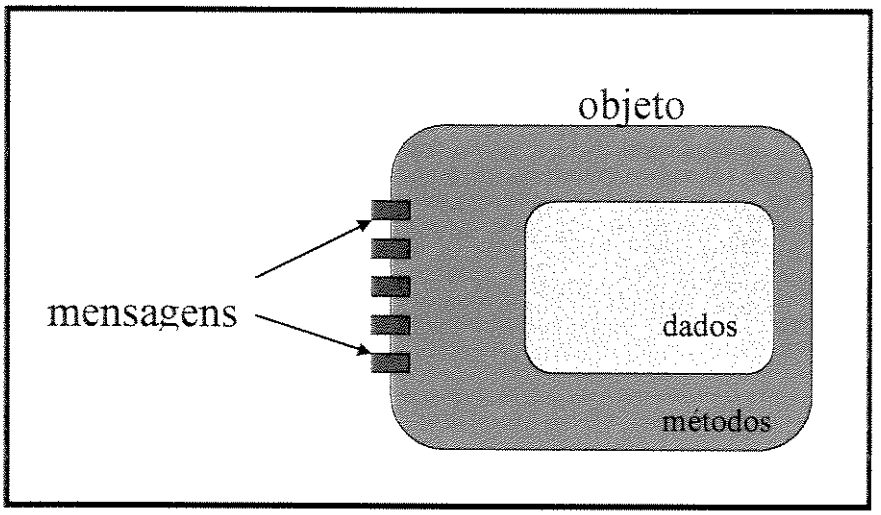

Figura 3.4: Anatomia de um objeto.

A programação orientada a objetos permite a reutilização do código através da *hereditariedade*. Uma classe-filha pode ser criada herdando as estruturas de dados e os métodos de uma classe-pai, assim novas classes podem ser criadas pela especialização da classe-pai, formando uma hierarquia de classes.

CAPÍTULO 4

PROPOSTA DE LINGUAGEM PARA MICROCONTROLADOR

Introdução

Neste capítulo é feita uma proposta de uma linguagem visual orientada a objetos para programação de microcontroladores batizada de O++. Esta proposta une os paradigmas da programação orientada a objetos e da programação visual. Este capítulo, inicialmente, fornece uma visão de como essa união pode gerar resultados positivos para o desenvolvimento de *software*. Também é fornecida uma visão da estrutura da linguagem proposta, ressaltando os aspectos de filosofia de programação e detalhes psicológicos levados em consideração em seu projeto.

4.1. Definições preliminares

4.1.1. Linguagens visuais híbridas

Na década de setenta, o pesquisador Allan Paivio [PAI71] fez retomar uma antiga tradição na psicologia, segundo a qual as imagens se constituem uma forma de representação fundamental de eventos na mente humana. Ele, na realidade, estabeleceu uma diferenciação dos tipos de informação guardadas em nossa memória. Sua teoria, chamada "teoria de dois códigos", mostra que empregamos dois sistemas de codificação distintos para representar o mundo em nossas mentes: um sistema figural e um sistema lingüístico. Ou seja, temos duas maneiras distintas para representar as informações: uma baseada em imagens e outra em palavras (figura 4.1).

Partindo dessa premissa, pode-se dizer que o uso de imagens (ícones, diagramas e gráficos, p. ex.) para representar algoritmos ou mesmo dados, aliado com descrições textuais, tendem a promover um melhor entendimento das aplicações (programas), visto que estamos trabalhando com dois tipos de códigos: textual e visual.

É nesse ponto que se alicerçam as linguagens visuais híbridas. Neste caso, o termo *híbrida* se refere a forma de representação das estruturas da linguagem. Esse tipo de linguagem faz uso tanto de figuras como de textos para expressar suas estruturas[ERW95]. As figuras ou imagens são utilizadas para possibilitar um enfoque visual-espacial das

estruturas inter-relacionadas, ou para realçar detalhes de estruturas individuais; enquanto os mecanismos textuais permitem uma melhor visão de estruturas lógico-seqüenciais e de recursão, como ilustra a figura 4.2.

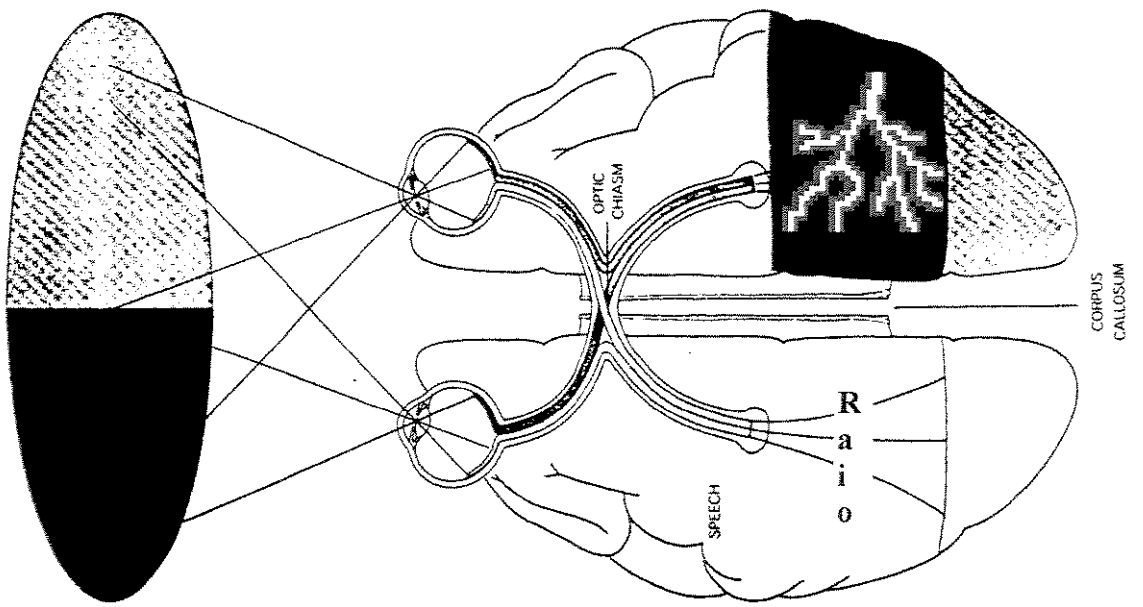

Figura 4.1: Os sistemas de codificação no cérebro

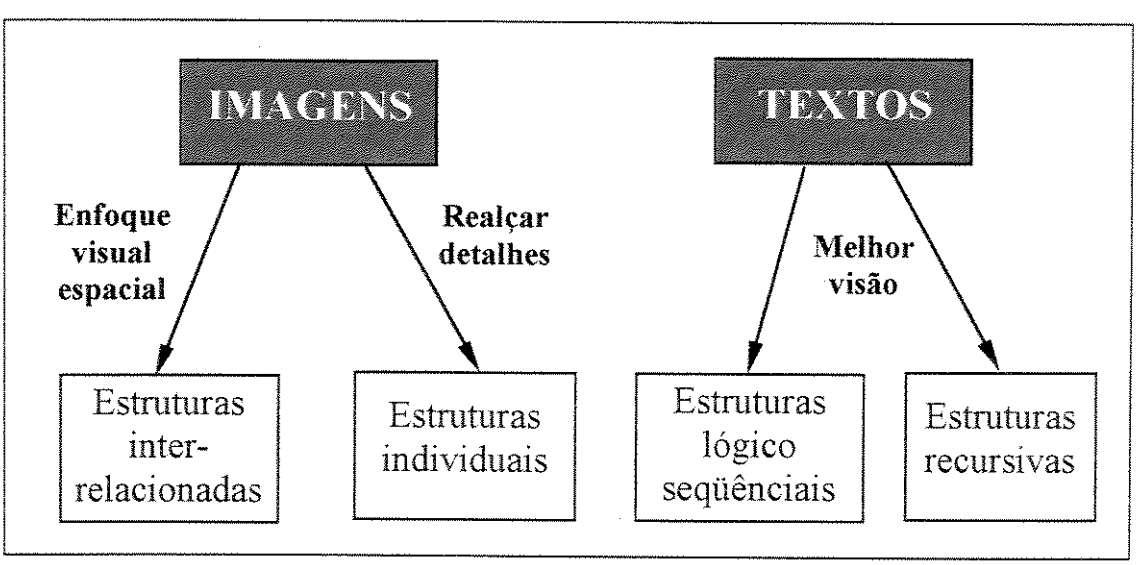

Figura 4.2: A combinação de texto e imagem nas linguagens visuais híbridas

A combinação desses enfoques não é tão recente. Consolidados trabalhos da década passada já fizeram uso desses dois mecanismos de representação (Pascal/HSD [DIA80] e Cartas NS [NAS84]).

4.1.2. Linguagens visuais orientadas a objetos

A *programação visual orientada a objetos* é um tipo de metodologia de programação que combina as técnicas de programação orientada a objetos com a programação visual (figura 4.3). Segundo Margaret Burnett[BUR95], este termo se aplica às linguagens orientadas a objetos que possuem uma sintaxe visual, como também se aplica às linguagens orientadas a objetos que possuem uma sintaxe textual, mas que tenham um ambiente visual de programação (ver ilustração na figura 4.4). Assim, é importante observar que um programa cuja saída seja gráfica e que tenha sido desenvolvido usando uma linguagem orientada a objetos não implica dizer que esta linguagem seja visual orientada a objetos. O termo "visual" implica no uso significativo de representações gráficas no processo de programação [SHU88].

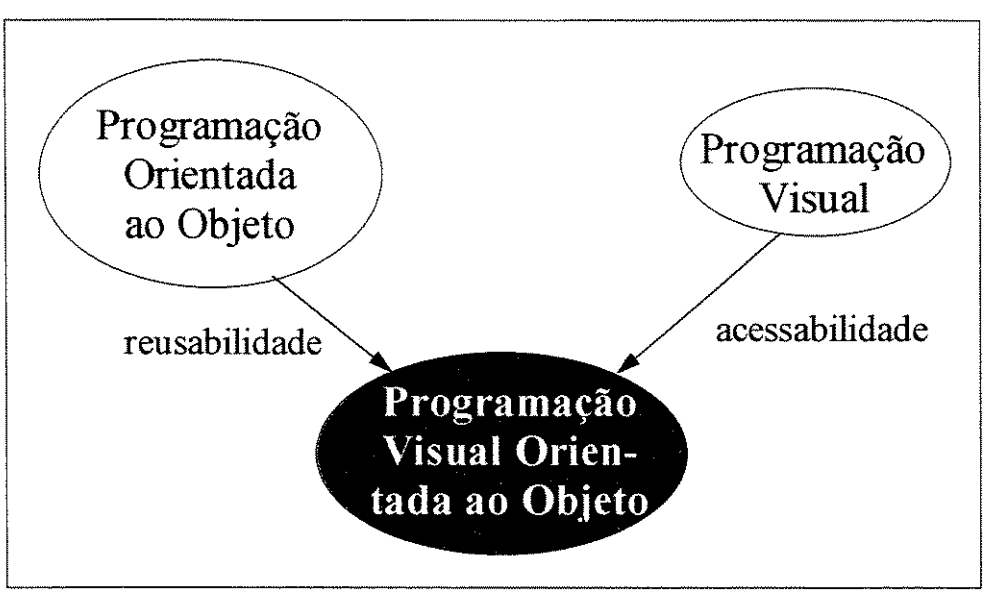

Figura 4.3: A idéia central da programação visual orientada a objetos

A **programação visual orientada a objetos** (Visual Object-Oriented Programming) é uma área emergente que combina as características da programação orientada a objetos

com técnicas de programação visual, como já foi mencionado. A meta da programação visual orientada a objetos é combinar as vantagens de cada enfoque. A reusabilidade e extensibilidade da tecnologia orientada a objetos e a acessibilidade da programação visual, com o objetivo de aumentar a produtividade dos desenvolvedores de *software* [TKA96].

Um dos principais motivos da "crise do *software*" é justamente a baixa produtividade dos desenvolvedores do mesmo, que tentam resolver os desafios do *software* desta década, usando antigas metodologias e ferramentas ultrapassadas[CHO97]. Assim, a junção da programação orientada a objetos com a programação visual se apresenta como uma nova alternativa para ajudar a superar esses desafios. A programação visual orientada a objetos permite melhorar a qualidade e aumentar a acessibilidade na troca de informação entre programadores e computadores, enquanto, ao mesmo tempo, permite a criação de programas que resolvam grandes e complexos problemas.

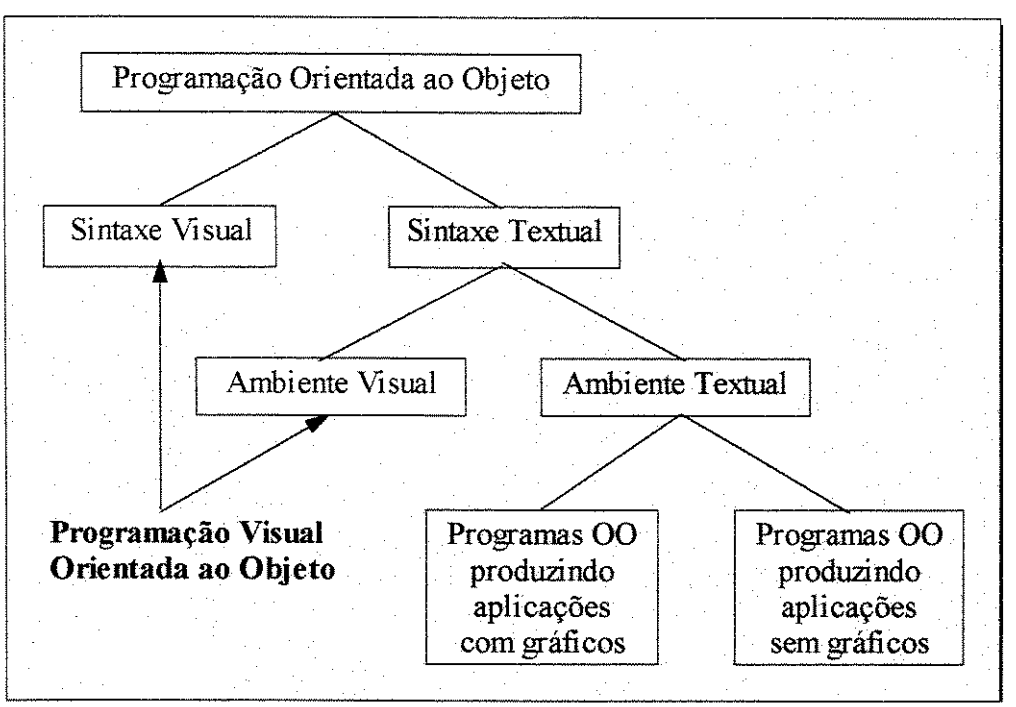

Figura 4.4: Classificação das linguagens orientadas a objetos

Por outro lado, mas ainda consolidando a programação visual orientada a objetos, a combinação da programação orientada a objetos com a programação visual tende a superar desafios e realçar qualidades de cada enfoque. A programação orientada a objetos, por exemplo, em se tratando de linguagens textuais, tem alguns desafios não superados, dentre

os quais, podemos destacar a falta de uma boa visão da hierarquia de classes, ou até mesmo, de sua própria anatomia (composição). Neste sentido, a programação orientada a objetos pode se servir da programação visual como meio proporcionador de melhor representação de suas estruturas, minimizando o impacto sentido pelos programadores tradicionais ao se depararem com a nova semântica da programação orientada a objetos. Isto, inclusive, já vem ocorrendo com vários produtos comercialmente disponíveis (ambientes gráficos de linguagem de programação orientada a objetos).

A possibilidade de descrever visualmente os componentes de uma classe (natureza dos dados e tipo de acesso), bem como representar graficamente a hierarquia das classes e o comportamento dos objetos, permite que se ampliem os alcances da programação orientada a objetos, proporcionando que um maior número de pessoas possa projetar e manter suas próprias aplicações. Estas pessoas podem não ser, necessariamente, especializadas em programação. Mesmo assim, podem montar suas aplicações, através da junção de partes (objetos) devidamente projetadas por especialistas, criando apenas seus objetos específicos no domínio de suas aplicações.

A programação visual também pode se servir da programação orientada a objetos, permitindo ampliar seu alcance, haja visto que a programação visual apenas consolidou seu uso em pequenas aplicações. Assim, usando objetos pré-fabricados, as linguagens visuais podem implementar aplicações maiores e permitir um melhor reuso de código de uma aplicação para outra.

4.2. A Linguagem O++

A linguagem que será proposta, batizada de O++[SOU98], é fruto da idéia que começou com o desenvolvimento de um ambiente gráfico para programação de microcontroladores, chamado ONAGRO [SOU95] [SOU96a] [SOU96b]. Apesar da nova linguagem não ter, em sua forma, nenhuma semelhança com a anterior, ela possui o mesmo domínio de aplicação e representa um estágio de nível mais alto para o desenvolvimento de aplicações microcontroladas.

4.2.1. Domínio de aplicação

O universo de aplicações que usam microcontroladores é bastante variado, conforme comentado no capítulo 1. Entretanto, todas as aplicações possuem um domínio comum. Elas manipulam sinais de entrada advindos de sensores e indicadores e geram sinais de saída para controlar tarefas e sinalizar condições, dentre outras operações. O processamento se transcorre com tomadas de decisão e cálculos usando esses sinais.

No projeto da linguagem O++, procurou-se estabelecer uma relação de proximidade entre o domínio da linguagem e o domínio da aplicação. As razões disso podem ser enquadradas sob dois pontos de vista complementares: a precisão da linguagem e suas características psicológicas.

Em primeiro lugar, deve-se salientar a necessidade de se projetar uma linguagem que fosse precisa e assim, tornou-se necessário evitar os problemas relacionados com a natureza da confusão dos símbolos gráficos. Desta forma, era condição indispensável que os elementos da linguagem fossem compatíveis com o "universo intelectual" das pessoas envolvidas no processo de programação. Então, a simbologia utilizada para representar dados e operações foi projetada seguindo o contexto das aplicações microcontroladas.

Em segundo lugar, e com maior impacto, deve-se salientar que uma linguagem de programação não pode obter sucesso sem que ela respeite as características psicológicas do programador. Ela deve ser projetada para ser usada pelo programador e não apenas para que a máquina possa executar o que foi programado. Nesse sentido, muito importante foi a compilação feita por Samuel P. Netto [NET87] a respeito das teorias modernas de psicologia da aprendizagem.

No livro, *Psicologia da Aprendizagem e do Ensino*, Samuel P. Netto relata, que "na segunda metade do século atual, novos modelos e concepções sobre a aprendizagem humana passaram a explorar mais os aspectos cognitivos desta, concentrando-se na investigação das formas pelas quais cada aprendiz seleciona, interpreta e transforma (processa) as informações em suas estruturas nervosas internas". Nesses modelos, a entrada de informação se desenvolve em duas etapas: de dentro para fora e de fora para dentro do organismo.

Segundo o autor, "a entrada refere-se tanto às portas sensoriais abertas, para que a informação passe do ambiente para o sistema nervoso do organismo, como aos mecanismos

e recursos de que este dispõe para buscar, detectar e captar parte da massa de informações disponíveis no meio que o cerca". A entrada de informação não ocorre em via única (fora para dentro ou dentro para fora) e sim numa transação que envolve duas etapas: primeiro, "o organismo é impelido à busca de um tipo determinado ou específico de informação que corresponde às suas intenções, necessidades e expectativas, aos seus propósitos e interesses (dentro p/ fora)"; segundo, "a informação captada apresenta características de estrutura, organização, intensidade e outras que a tornam mais saliente, facilitando, portanto, sua captação, ou que reduzem consideravelmente sua possibilidade de ser objeto de atenção, na competição com outras informações disponíveis (fora para dentro)". Como exemplos, o autor lembra o emprego da camuflagem, usada para esconder alguma coisa, e dos comerciais de TV, que realçam detalhes do produto que se quer vender.

Seguindo esse modelo de aprendizagem, o projeto da linguagem O++ procurou seguir as necessidades e os interesses das pessoas envolvidas no desenvolvimento de *software* para microcontroladores, uma vez que o domínio da linguagem é um subconjunto do domínio das aplicações microcontroladas e seus elementos apresentam características gráficas que tornam os algoritmos mais palpáveis para seus desenvolvedores.

4.2.2. A idéia de se utilizar a programação orientada a objetos

Uma outra idéia importante, incorporada à linguagem O++, foi o uso do paradigma da programação orientada a objetos. Em síntese, este paradigma permite um aumento de produtividade pelo reuso de elementos de *software*. Ele também contempla as características psicológicas do programador, pois nos mesmos modelos de aprendizagem, comentados anteriormente [NET87], "a organização da informação entrante conduz à aprendizagem mais rápida e à melhor memorização. A recuperação da informação armazenada se torna muito mais fácil se esta é organizada desde o início. Os itens na memória a longo prazo são tanto mais prontamente recuperados quanto mais se acham estruturados ou **categorizados**. O estratagema dos métodos propostos para o desenvolvimento de sistemas de memória consiste em aprender a organizar as coisas que devemos aprender, de modo que eles possam ser novamente encontrados em nossa memória, quando necessário".

Esse ponto vai ao encontro da estruturação da programação orientada a objetos. Os objetos (entidades) são organizados ou categorizados em classes, formando uma estrutura hierarquizada. Assim, possibilitando uma boa organização da informação que é captada pelas pessoas envolvidas no desenvolvimento dos programas.

Neste sentido, a linguagem O++ foi projetada à luz dos conceitos da programação orientada a objetos. Ela usa um subconjunto de declarações textuais em C++, que é orientada a objetos; introduz uma simbologia gráfica para representar classes e faz uso de cartas NS para descrever os algoritmos, como ilustra a figura 4.5.

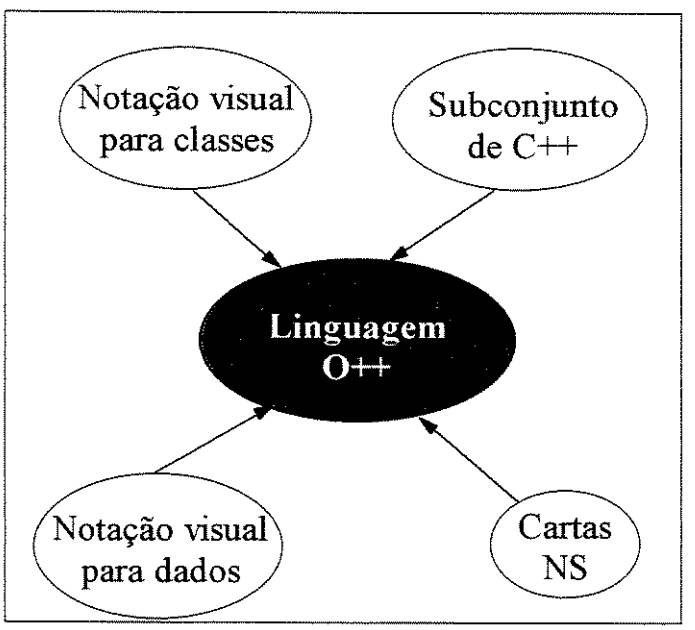

Figura 4.5: A idéia central da linguagem O++

Apesar da linguagem O++ permitir o uso de conceitos orientados a objetos, é possível a implementação de aplicações usando estruturas puramente procedimentais. A linguagem O++ não exige que as aplicações sejam implementadas usando a programação a objetos, ou seja, pode-se implementar aplicações usando a programação procedimental. Isto se deve ao fato de muitos programadores desse tipo de aplicação não ter (ou não querer ter) conhecimento sobre a programação orientada a objetos. Assim, se amplia o universo de programadores que podem utilizar a linguagem O++, mesmo que não sejam exploradas todas as suas potencialidades.

4.2.3. O modelo de solução de problemas

Basicamente, os programas em O++ são constituídos pela junção de dois modelos de solução de problema: um modelo estático e um modelo dinâmico, como mostra a figura 4.6.

No modelo estático se trabalha com o relacionamento hierárquico das classes e com a descrição de suas estruturas, dando ênfase aos atributos que cada classe deve possuir.

Já no modelo dinâmico, enfoca-se, primordialmente, o estabelecimento de troca de mensagens entre objetos e como cada objeto deve se comportar ao receber uma mensagem. Para isso, é feito um detalhamento dos métodos ou funções-membros.

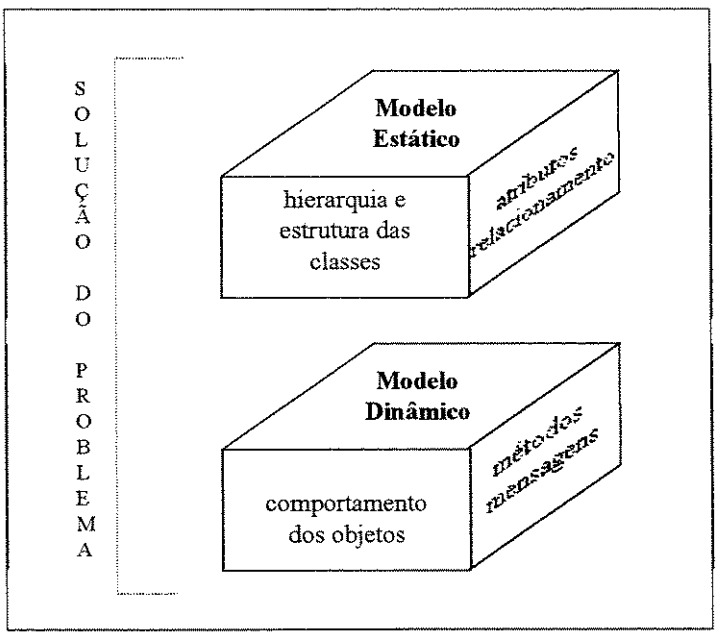

Figura 4.6: Modelamento de solução de problemas

4.2.4. Hierarquia e descrição de classes

A linguagem O++ se utiliza de um diagrama icônico para representar a hierarquia de classes. Cada classe tem um nome (texto) e uma figura (ícone) associados. Este ícone é escolhido ou desenhado de forma a expressar, da melhor forma possível, os objetos representados por aquela classe. A hierarquia é especificada pelos níveis de cima para baixo (top-down), como mostra a figura 4.7a. Nesta figura, podemos observar que a classe D é

derivada da classe B, que por sua vez é derivada da classe A. Evidentemente, os efeitos da derivação são os mesmos de qualquer linguagem orientada a objetos. A classe derivada herda as estruturas da classe de nível mais alto (classe pai).

Os diagramas oferecem ao projetista uma representação que se mostra mais compreensível do que a descrição lingüística. Não porque a descrição lingüística possa apresentar lacunas, mas porque os diagramas permitem uma captação mais fácil de informações. As conseqüências das interações dos componentes em um diagrama podem, para muitas pessoas, ser instantaneamente óbvia. Isso porque os diagramas permitem que o projetista retire informações sem utilizar inferências lingüísticas, fazendo apenas uso de habilidades inerentes ao cérebro humano, como a inferência por estruturas espaciais.[ADD97]

O diagrama hierárquico, em contraste com a descrição textual, evidencia de forma explícita a relação entre as classes. Por exemplo, podemos ver diretamente na figura 4.7b, que a classe J é herdeira da classe C, sem ter que realizarmos qualquer referência as relações intermediárias, mas simplesmente observando que as classes J e C fazem parte de um mesmo grupo de percepção. Em outras palavras, elas fazem parte de uma mesma linhagem.[OLI97]

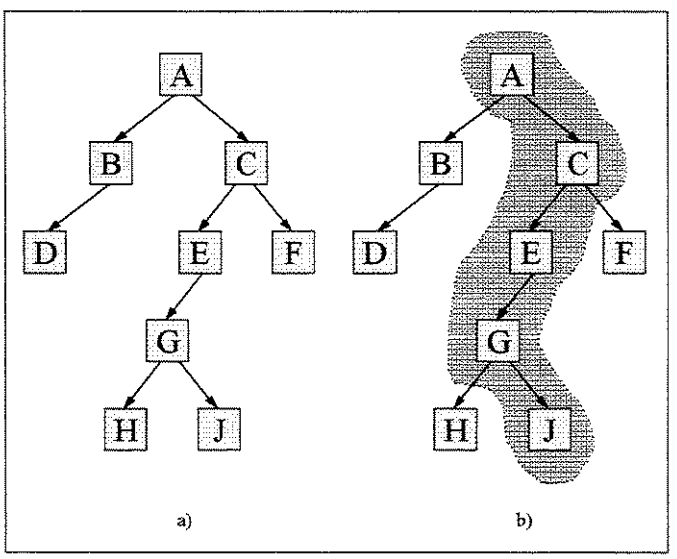

Figura 4.7: Exemplo de hierarquia de classes

Como já mencionado, a relação entre as classes é expressa de forma visual, ao passo que a identificação é feita tanto por meio de informação gráfica (um ícone), como por meio de informação textual (um nome). De forma semelhante, a informação textual é usada para identificar as variáveis-membros (membros de dado) e os métodos pertencentes à classe. Os métodos, por sua vez, são especificados por meio de Cartas NS e declarações em C++, conforme será discutido no item *Estruturas de Controle*.

Para descrever as classe na linguagem O++ foi utilizada uma notação inspirada na proposta de Wasserman [WAS90]. Toda classe é descrita tomando-se por base um retângulo com o nome da classe. Este retângulo delimita a área de descrição ou declaração da classe. Adicionalmente, introduzimos a possibilidade de se associar um ícone a cada classe.

Dentro desse retângulo são introduzidos os atributos (membros de dado e funções membros) que compõem a classe. Estes atributos podem ser protegidos ou públicos. Os elementos protegidos só são acessíveis pela classe ou por seus descendentes; enquanto os elementos públicos podem ser manipulados fora da classe.

A distinção entre protegido e público é feita de forma visual. O elemento que esteja totalmente contido dentro do retângulo é considerado protegido, já os elementos que estejam na fronteira do retângulo são considerados públicos. Veja exemplo na figura 4.8.

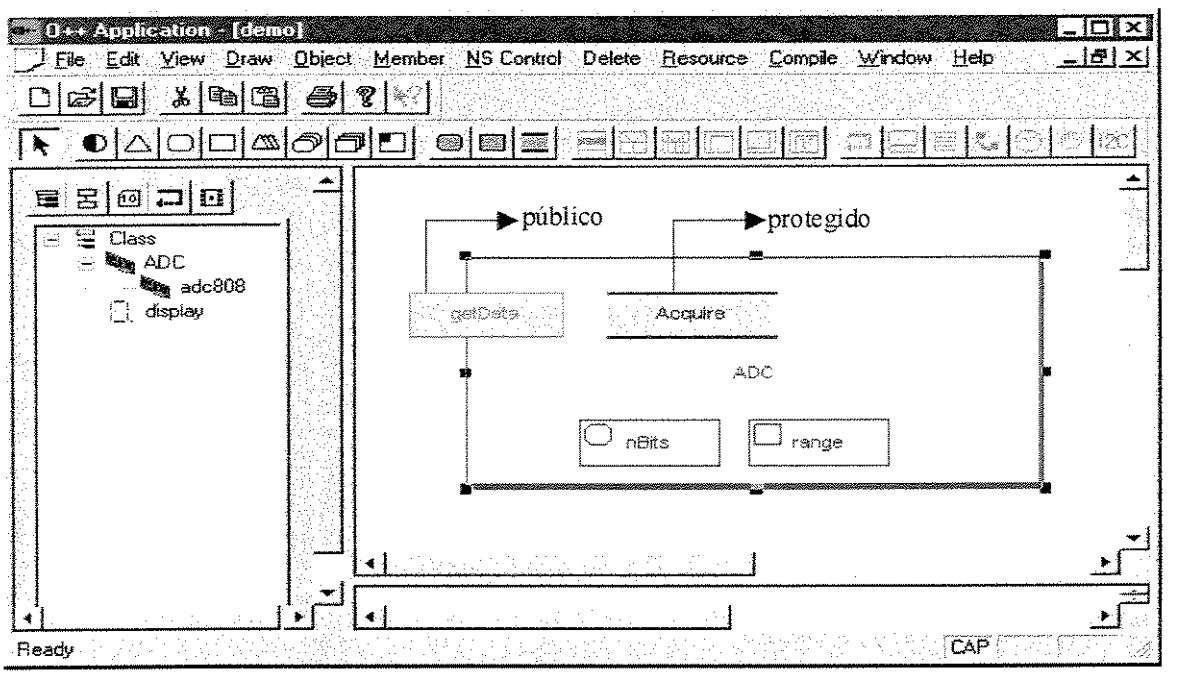

Figura 4.8: Exemplo de descrição de classe

Pelo exposto, pode-se observar que a linguagem O++ não é uma linguagem orientada a objetos pura. Ela permite o encapsulamento de dados e algoritmos e o ocultamento de informação, porém este ocultamento não é plenamente garantido, haja visto a possibilidade permitida pela linguagem de se criar "furos" no escudo de proteção das informações, através da criação de membros públicos. Por outro lado, quando ocorre proteção de dados, esses "furos" podem evitar uma excessiva troca de mensagens entre objetos, diminuindo o "overhead" de processamento.

4.2.5. Tipologia e representação de dados

A representação dos dados combina a informação textual usada para nomear um membro de dado ou uma função membro com a simbologia gráfica usada para representar o tipo de dado do membro ou do valor de retorno da função. Assim, além da simbologia gráfica, cada membro de dado ou função membro possui um nome que é inserido dentro de sua representação gráfica.

Adotamos parcialmente a simbologia sugerida por Calloni & Bagert [CAL94] e introduzimos algumas modificações. Na notação resultante, o dado do tipo inteiro é representado por um retângulo com os cantos arredondados (sugerindo arredondamento) e o dado do tipo real é representado por um retângulo com os cantos aguçados, sugerindo precisão (veja figura 4.9). Esta simbologia tem sua coerência ressaltada quando observamos os tipos de dados como conjuntos: o retângulo com cantos arredondados está contido no retângulo com cantos aguçados, assim como o conjunto dos números inteiros está contido no conjunto dos números reais.

Os dados que armazenam valores lógicos são representados por uma circunferência parcialmente preenchida, sugerindo os valores **verdadeiro** e **falso**. Já os dados que armazenam caracteres são representados por um triângulo, em alusão a **letra** grega delta.

Os membros de classe (membros comuns a todos os objetos de uma classe) são representados por retângulos de linha dupla, a fim de diferenciá-los dos membros de instância.

De forma intuitiva, a simbologia sugere que as estruturas homogêneas (*arrays*) sejam representadas por uma coleção ou conjunto de elementos do tipo base.

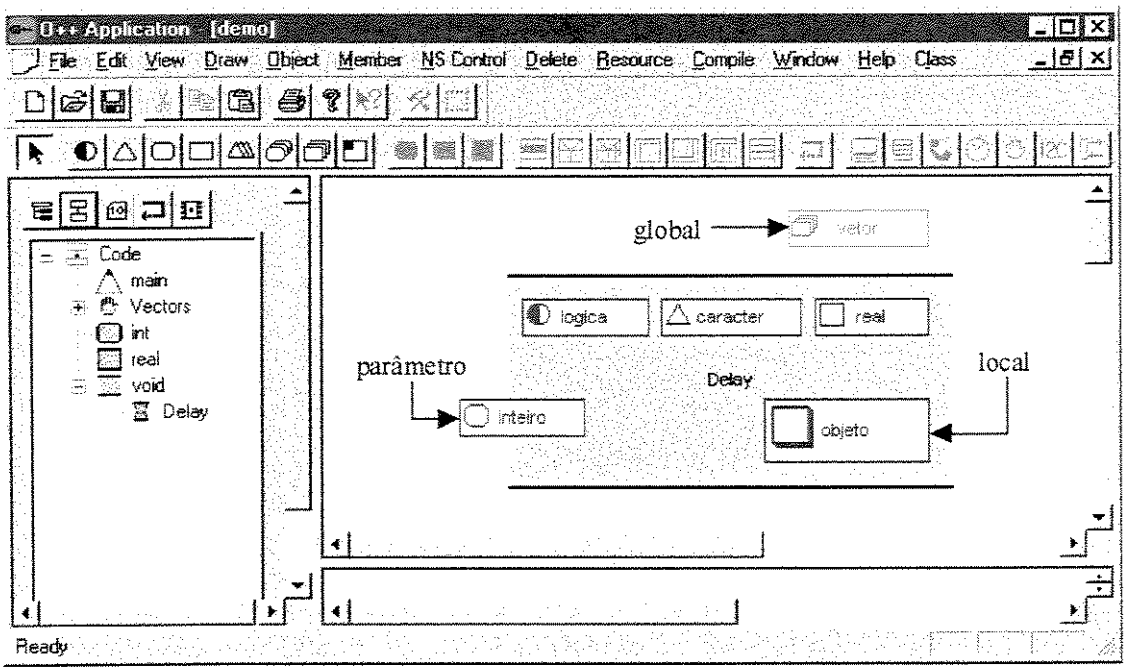

Figura 4.9: Tipos de dados

Quando o elemento a ser representado é um objeto, a notação faz uso do ícone de identificação da classe a qual o objeto pertence e de um identificador para o próprio objeto.

A distinção entre variáveis globais, locais e parâmetros se faz utilizando a **relação espacial** entre o retângulo delimitador da variável com o retângulo delimitador do módulo no qual a variável é definida. Assim, se a variável está totalmente inserida no módulo, diz-se que ela é local. Já se ela está totalmente fora do módulo, diz-se que ela é global. Finalmente, se ela se encontra na fronteira do módulo, ela é considerada como um parâmetro de entrada do módulo.

4.2.6. Estruturas de controle

A linguagem O++ prove suporte à programação estruturada baseado na técnica de arranjos gráficos proposta por Nassi-Shneiderman[NAS84]. Os algoritmos são representados graficamente, através das estruturas mostradas na figura 4.10. Esta notação permite a construção de programas bem estruturados, substituindo os desvios incondicionais (GOTO's) por arranjos embutidos.

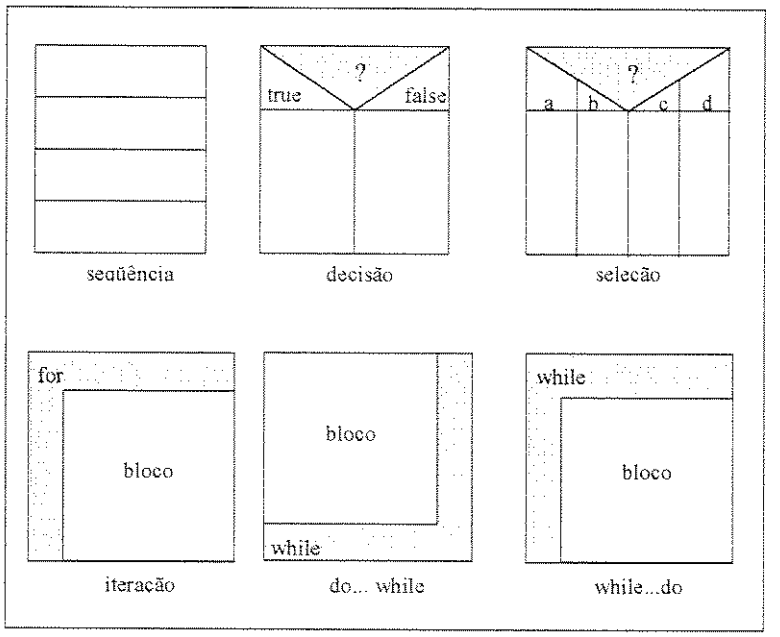

Figura 4.10: Estruturas de controle NS

Este tipo de diagrama usa estruturas visuais para representar o fluxo de controle e informações textuais para descrever os dados manipulados, testes de condições, indicação de número de repetições, dentre outras.

As declarações básicas da linguagem O++ que são inseridas textualmente nos diagramas são um subconjunto de declarações em C++. Portanto, procurou-se dar a linguagem O++, no tocante às estruturas textuais, o mesmo escopo de aplicação da linguagem base C++. Assim, as chamadas de funções, as trocas de mensagens, a implementação de recursividade e de polimorfismo e as referências aos objetos são feitas textualmente da mesma forma que na linguagem C++.

4.2.7. Operações dedicadas

Além das estruturas básicas de controle, a linguagem O++ dispõe de operações voltadas ao domínio das aplicações microcontroladas (veja figura 4.11). Dentre essas operações, podemos destacar o manuseio gráfico dos recursos do microcontrolador, através da manipulação das portas de comunicação, do sistema de interrupção e dos contadores e temporizadores. Além disso, podemos também destacar na linguagem O++, o conjunto de operações específicas para manipular *arrays*.

As operações para manipulação das portas de comunicação (serial e paralela) permitem a programação dos dispositivos internos que implementam as interfaces e o acesso de leitura e escrita nessas portas.

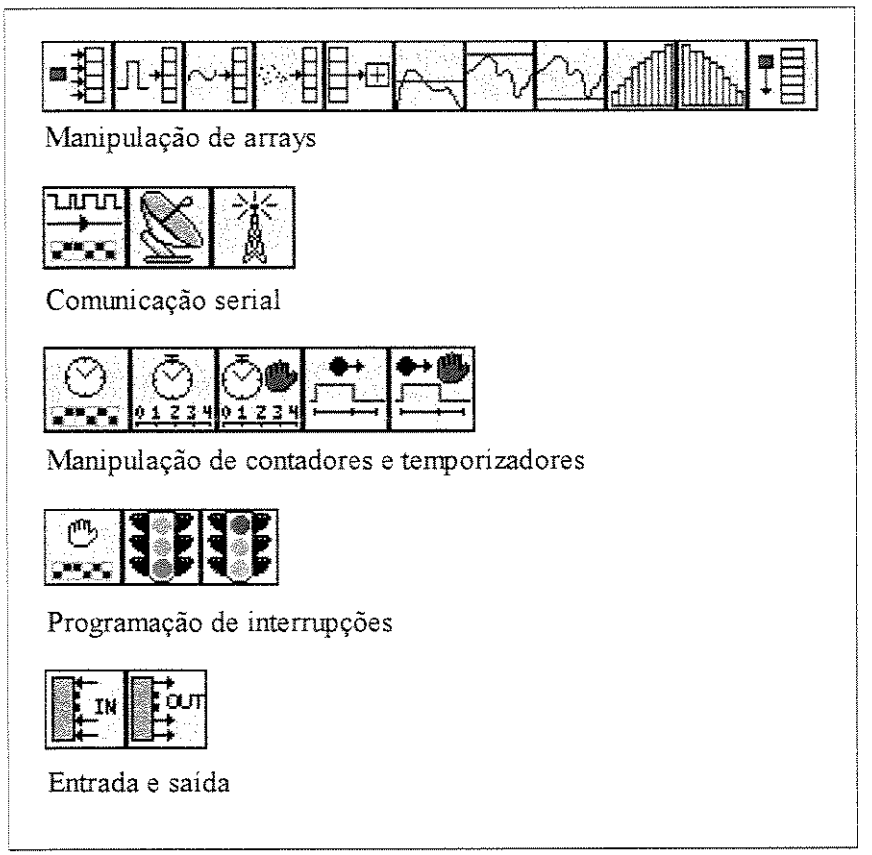

Figura 4.11: Operações dedicadas

O sistema de interrupção possui operações que podem programá-lo em qualquer trecho de código executável do programa. Também permite que seja habilitada ou desabilitada qualquer fonte de interrupção mascarada.

As operações que manipulam os contadores e temporizadores permitem, além da geração de código de iniciação desses dispositivos, a contagem de eventos e a geração de pulsos de forma bastante direta para o programador.

Finalmente, as operações com *arrays* envolvem: iniciação de todos os elementos com um certo valor; iniciação do *array* com valores representando um pulso quadrado; iniciação do *array* com valores representando um pulso senoidal; iniciação do *array* com valores aleatórios; cálculo da soma dos valores do *array*; cálculo da média dos valores do

array; cálculo do valor máximo do *array*; cálculo do valor mínimo do *array*; ordenação ascendente do *array*; ordenação descendente do *array* e pesquisa de um elemento do *array*.

Essas operações dedicadas são análogas aos códigos de operação de uma instrução de máquina, ou seja, elas essencialmente indicam a natureza do processamento que se deseja realizar. É preciso, em complemento, que se indique os parâmetros que serão usados na operação. Esses parâmetros normalmente são constantes, identificadores de variáveis e de portas de e/s, como pode ser visto na figura 4.12. Nesta figura, a última operação, por exemplo, representa a saída do valor da variável *med* em uma porta paralela chamada *display*. Esses identificadores foram definidos em locais apropriados e facilmente resgatáveis, conforme será comentado no próximo capítulo.

Para representar as operações dedicadas, adotou-se o uso de ícones. Durante a elaboração dos ícones houve a preocupação em torná-los o mais intuitivo possível. Assim, eles foram organizados em classes de operações, conforme mostra a figura 4.13, onde cada classe agrupa todos os ícones que representam operações com características comuns. São disponíveis, por exemplo, três operações para manipular o sistema de interrupção (configurar, habilitar e inibir interrupções) agrupadas numa mesma classe.

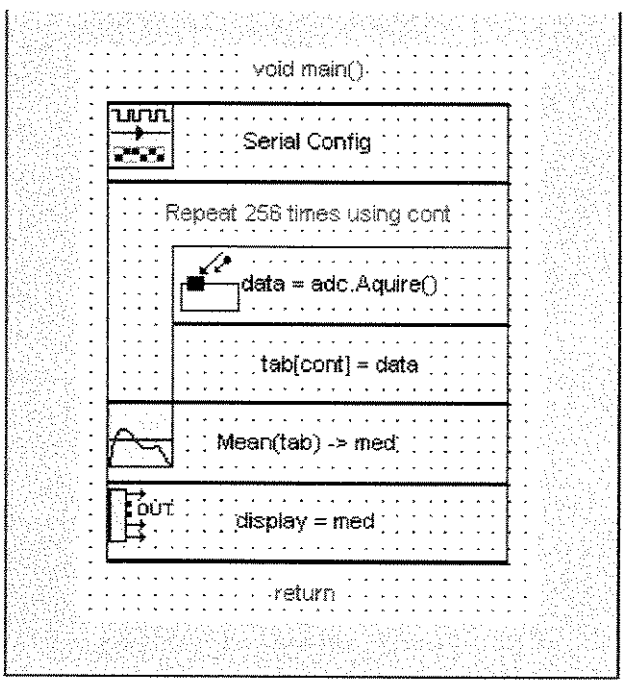

Figura 4.12: Exemplo de operações em um módulo em O++

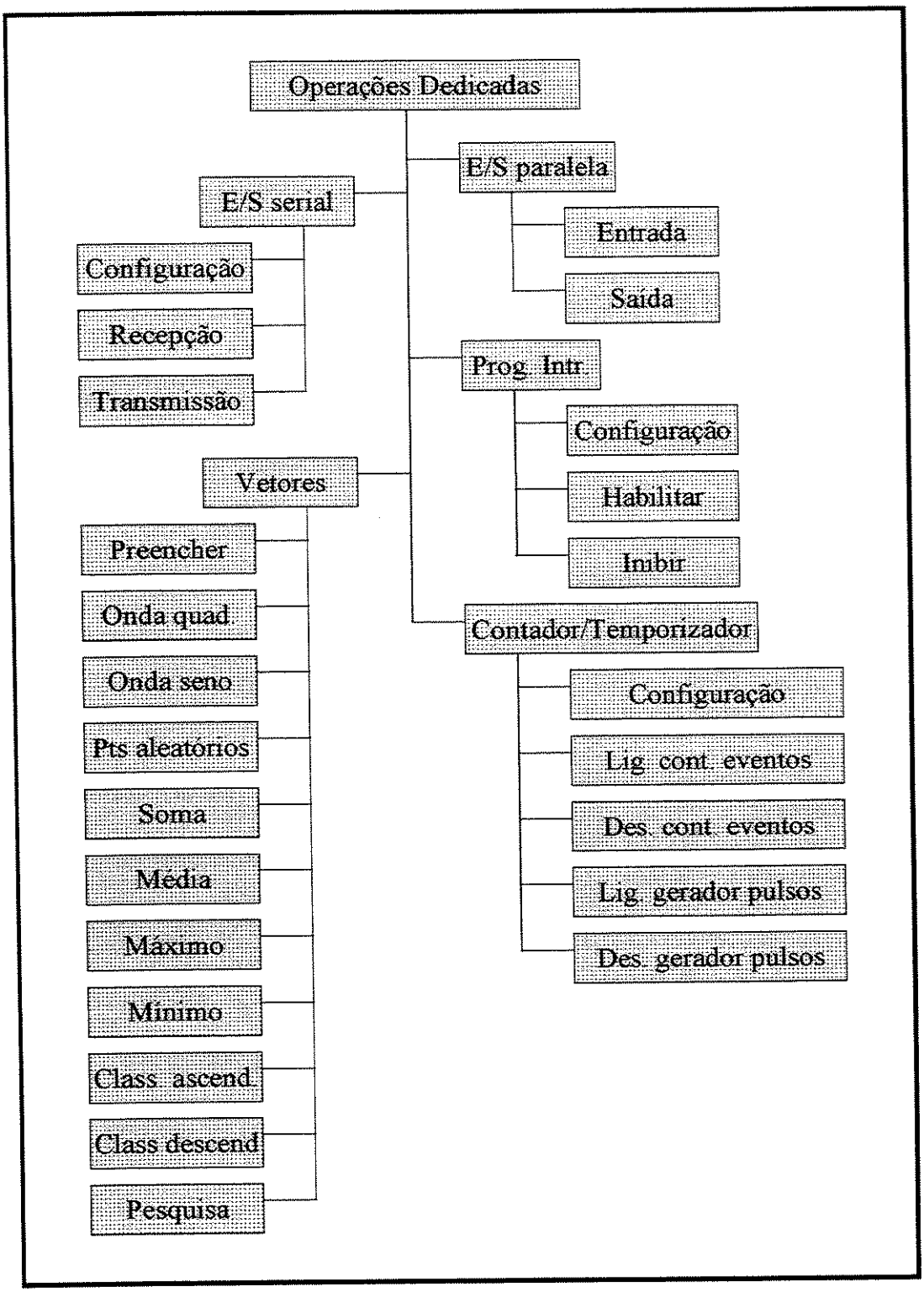

Figura 4.13: Hierarquia dos ícones de operação

É importante ressaltar que mesmo havendo diferentes ícones numa classe, eles têm características visuais semelhantes. A simbologia criada procurou ressaltar o que eles têm de peculiar, como pode ser observado na figura 4.11. Os últimos dois quadros representam ícones diferentes relacionados com a operação de entrada e saída em porta paralela. Existem pequenas, mas importantes, mudanças no desenho a fim de diferenciá-los e, em contrapartida, existe ao mesmo tempo entre eles uma semelhança que nos leva a perceber que eles pertencem a mesma classe.

CAPÍTULO 5

RESULTADOS OBTIDOS

Introdução

Este capítulo mostra os resultados obtidos no trabalho, relatando-se o desenvolvimento do protótipo O++. Inicialmente, é feita uma descrição do ambiente O++ do ponto de vista operacional, enfocando-se os aspectos de utilização do ambiente. É dada uma descrição dos mecanismos utilizados na interface com o usuário, bem como das operações implementadas especificamente para microcontroladores. O item seguinte, o foco é a implementação do compilador. É abordado o modelamento da linguagem, a estrutura do código e os mecanismos de tradução usados, a geração do código em linguagem C e o mecanismo utilizado para implementar a biblioteca de funções.

5.1. O ambiente de programação O++

5.1.1. A janela principal do ambiente

Conforme ilustra a figura 5.1, o ambiente O++ foi estruturado para trabalhar com três vistas conjuntas. A primeira, chamada de **vista de catálogo**, é usada para organizar os elementos do programa (classes, referências ao código, dados e elementos de *hardware*). A segunda, chamada de **vista de *script***, é usada para desenho de código, descrição de classes e descrição de dados. E a última, chamada de **vista de saída**, é usada para exibir mensagens de erro e/ou advertências ao usuário. Além disso, o ambiente O++ possui uma barra de ferramentas específicas, que acelera a escolha das tarefas que o usuário deseja realizar.

A vista de catálogo, ilustrada pela figuras 5.2 e 5.3, é composta por cinco árvores, que armazenam informações sobre a hierarquia das classes; as subrotinas definidas pelo usuário; os dados relativos a um módulo (uma função membro ou uma subrotina); os possíveis módulos que podem ser ativados; e algumas informações relativas ao *hardware* do microcontrolador.

A árvore que armazena a hierarquia das classes (figura 5.2a) é montada diretamente pelo usuário, seguindo a estruturação das classes usadas na aplicação. As outras árvores são

montadas seguindo uma estruturação própria, que é pertinente ao tipo de informação que ela armazena.

A árvore que armazena as subrotinas (figura 5.2b) é estruturada em cinco ramos. O primeiro é formado apenas pelo módulo de partida do programa, chamado de ***main***; o segundo corresponde aos vetores de interrupção, identificado pelo rótulo ***Vectors***; os últimos três ramos estão relacionados com o tipo de retorno da subrotina (***int***, ***real*** e ***void***).

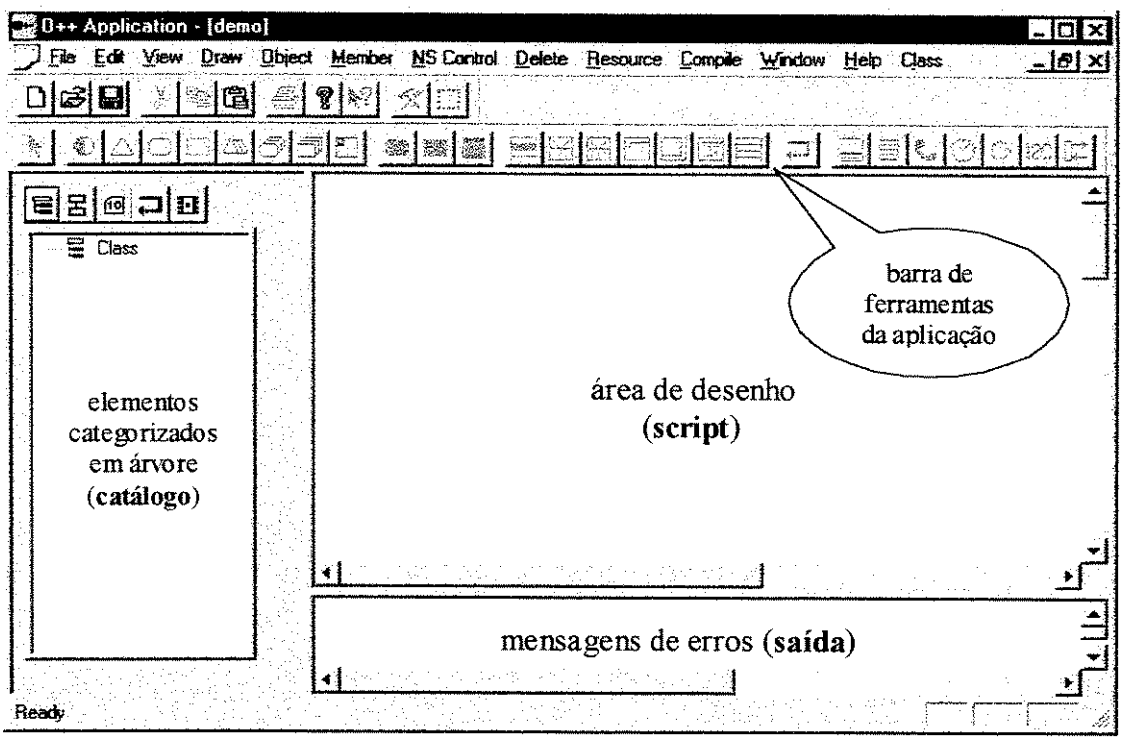

Figura 5.1: A vista principal do ambiente O++

A árvore que armazena os dados (figura 5.2c) é estruturada de acordo com o escopo de seus elementos. Há quatro ramos: o primeiro está relacionado com os dados globais; o segundo indica os dados localmente definidos; o terceiro corresponde aos dados advindos de parâmetros de entrada; e o quarto ramo armazena os membros de dados dos objetos acessíveis pelo módulo em foco.

A árvore de chamada de módulo (figura 5.3a) é organizada em duas seções. A primeira está relacionada com as mensagens que podem ser ativadas e a segunda é formada por funções ou subrotinas que podem ser chamadas. As mensagens, por sua vez, são categorizadas seguindo o escopo dos objetos acessíveis ao módulo selecionado e que

pertencem à classe herdeira da mensagem; já as subrotinas são organizadas seguindo o valor de retorno.

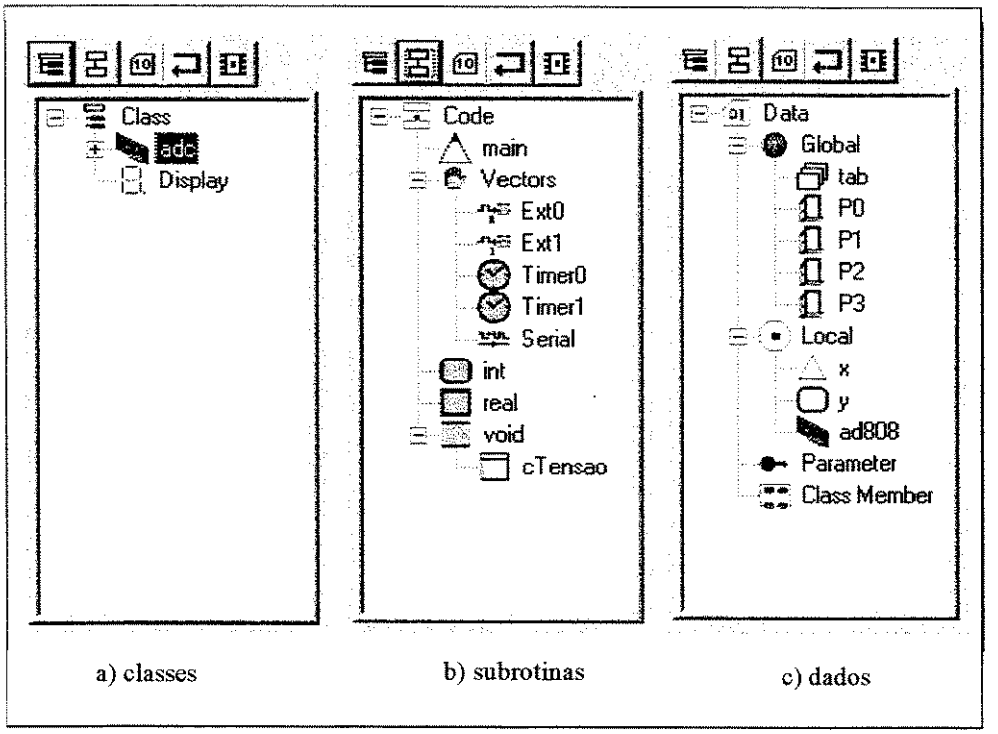

Figura 5.2: As árvores de classes, subrotinas e dados

Finalmente, a árvore que armazena os elementos do *hardware* do microcontrolador (figura 5.3b) é, nesta versão inicial, estruturada apenas levando-se em conta as portas de e/s pré-definidas e as definidas pelo usuário. A idéia é, futuramente, introduzir novos elementos do *hardware* do microcontrolador (conversores A/D e D/A, WATCHDOG, gerador de PWM, dentre outros dispositivos) que possam ser acessados pelas instruções da linguagem.

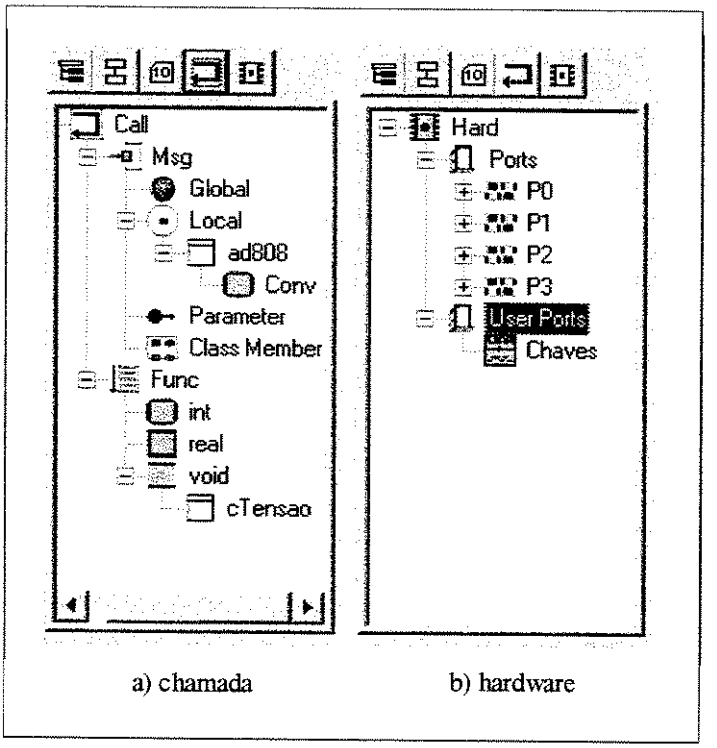

Figura 5.3: As árvores de chamada e de *hardware*

5.1.2. A interface homem-máquina

Para promover facilidade de uso e reduzir a taxa de erro do usuário, a interface homem-máquina, no ambiente O++, utiliza dois importantes mecanismos: a visualização automatizada dos elementos da vista de catálogo e a troca de informações entre as vistas do ambiente, por meio do recurso DRAG-DROP.

O ambiente O++ ativa, automaticamente, a visualização das árvores de acordo com o tipo de desenho que o usuário está realizando na vista de *script*. Essas árvores também podem ser ativadas, diretamente pelo usuário, através dos botões localizados acima da estrutura de árvore, conforme mostram as figuras 5.2 e 5.3.

A montagem das árvores de dados (figura 5.2c) e de módulos (figura 5.2b) é feita, automaticamente, seguindo o contexto da operação selecionada e do módulo ativo. Ao se selecionar uma operação, o ambiente O++ busca os elementos de dados definidos no escopo do módulo ao qual pertence a operação, monta a árvore de dados e realiza sua visualização. Quando a operação selecionada for de ativação de mensagens ou de chamada de subrotina,

o ambiente O++ monta e visualiza a árvore de subrotinas e funções membros, visíveis pelo módulo ao qual pertence a operação.

Assim, a interface homem-máquina altera a disposição das informações disponíveis, de acordo com o contexto da operação em foco. Isto, é claro, agiliza o desenvolvimento, pois diminui o tempo de construção dos programas e facilita a diminuição de erros do usuário, haja visto que a organização das informações segue o escopo do módulo que está sendo construído. As operações de um módulo só enxergam as informações definidas para aquele módulo, não sendo possível, por exemplo, o uso de um dado definido fora do alcance desse módulo.

Esse esquema de organização de informações oferece, ainda, duas importantes vantagens: a primeira se refere a rápida recordação de nome, tipo e escopo dos elementos do programa; e a segunda consiste no forte vínculo de uma operação com os elementos de programa (um dado, por exemplo) que ela manipula, ou seja, as operações não trabalham com os nomes dos elementos, mas sim com os objetos que implementam esses elementos. Desta forma, qualquer mudança de nome, feita a um elemento do programa, é, automaticamente, atualizado em todas as operações que referenciam esse elemento, evitando assim, a propagação de erros sintáticos de checagem de nomes.

Uma vez definido um elemento do programa (classes, membros, portas de e/s, etc), é feita a armazenagem deste elemento na correspondente árvore da vista de catálogo. Então, quando se cria uma nova classe, ela é inserida na árvore de classes. Da mesma forma, quando um novo membro de dados é criado, a árvore de dados armazena tal elemento e assim para os demais tipos de elementos do programa.

Os elementos do programa, armazenados na vista de catálogo, necessários para realizar uma operação, podem ser arrastados (DRAG) da respectiva árvore e depositados (DROP) dentro da imagem que representa a operação, contida na vista de script. Com isso, o processo de descrição das operações é bastante agilizado, dispensando a abertura de janelas de diálogo de descrição ou a digitação de nomes dos elementos manipulados pelas operações.

5.1.2.1. Sinalizador de pendências

No ambiente O++, todo elemento visual possui um *sinalizador de pendência*, que serve para indicar, visualmente, que falta alguma informação para completar a descrição do elemento. O sinalizador de pendência é muito útil quando o programador deseja introduzir os elementos do programa, sem ter que detalha-los no momento da inserção, mas sim, posteriormente, em outra fase da programação. Assim, o ambiente O++ sinaliza que determinados elementos não tiveram todas as descrições concluídas, tornando muito fácil a detecção dos elementos que ainda não estejam prontos.

O sinalizador de pendência trabalha com dois níveis de sinalização: um para indicar pendências de alto nível de relevância (pendências de informações imprescindíveis à compilação do programa); e outro nível para indicar pendências de baixo nível de relevância (as que podem ser assumidas por *default* e que não afetam a compilação do programa). A diferenciação entre os dois níveis de pendência é feita pela cor do símbolo de pendência (uma interrogação). A cor vermelha é usada para indicar as pendências de alto nível e a verde para indicar as de baixo nível de relevância.

5.1.2.2. Geração de mensagens e a passagem de parâmetros

A geração de mensagens e a simples chamada de subrotina ocorre por intermédio da operação de ativação de módulo, representada pelos ícones mostrados na figura 5.4. Como relatado anteriormente, quando essa operação recebe o foco de entrada, a vista de catálogo torna visível a árvore de subrotinas e funções membros acessíveis ao módulo ao qual essa operação pertence. Através do recurso DRAG-DROP, o programador completa a operação, selecionando o módulo que se quer ativar.

Através de duplo clique do *mouse*, ele pode visualizar, na vista de *script*, o arcabouço do módulo que está sendo referenciado. Este arcabouço diz respeito aos parâmetros de entrada, conforme ilustra o exemplo da figura 5.5 (o módulo ativado possui dois parâmetros: p1 e p2, do tipo inteiro e real, respectivamente).

Nesse momento, a vista de catálogo torna visível os elementos de dados do módulo origem, para que o programador, através do DRAG-DROP, selecione os parâmetros que serão passados ao módulo que será ativado.

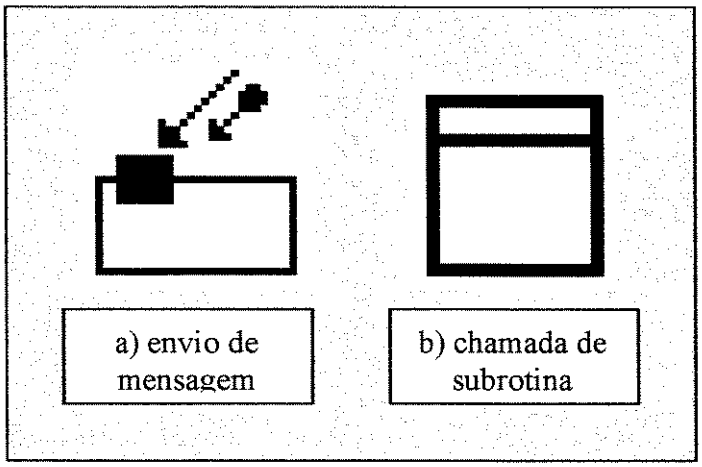

Figura 5.4: Ícones que representam ativação de módulo

Portanto, a definição da passagem de parâmetros é realizada de forma bastante didática, pois o programador, através da passagem física dos elementos de dados envolvidos, define quais os parâmetros que serão passados, usando o recurso DRAG-DROP.

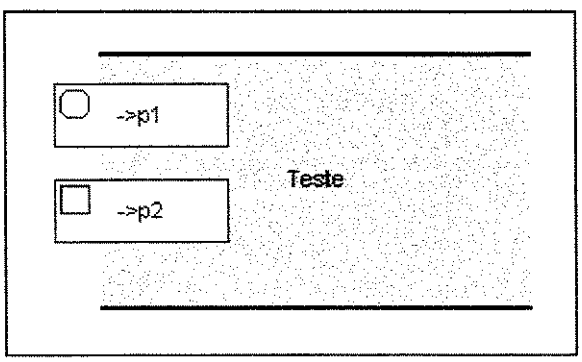

Figura 5.5: Módulo exemplo, com dois parâmetros

5.1.2.3. Configuração dos recursos do microcontrolador

Como se sabe, a maioria dos microcontroladores possuem além da CPU, uma série de componentes integrados num mesmo *chip*. Dentre esses componentes, podemos citar: o controlador de interrupção, o canal de comunicação serial, os temporizadores/contadores e as portas de I/O em geral; que são os mais usuais. Estes componentes, chamados de recursos do microcontrolador, possuem características peculiares, podendo mudar de uma

família para outra, tornando difícil realizar a programação ou configuração de tais elementos de forma padronizada.

Para configurar esses recursos, o usuário da linguagem O++ deve usar funções personalizadas, introduzindo o código necessário para utilizar adequadamente os recursos desejados. Entretanto, a linguagem oferece a possibilidade de configuração desses recursos, através de caixas de diálogos, para uma única família de microcontroladores. No caso, a família escolhida foi a MCS-51 da INTEL, que é atualmente uma das mais difundidas no mercado.

O objetivo desses diálogos é facilitar a configuração dos referidos recursos, permitindo ao usuário, uma programação em alto-nível dos recursos que serão utilizados na aplicação. As informações solicitadas pelos diálogos geram uma maior abstração de detalhes de configuração, como os *bits* que indicam o modo de trabalho de um certo recurso; nomes de registros ou *bits* de registros usados na configuração; endereços de memória associados; flags e máscaras; dentre outros.

Esses diálogos podem ser abertos de duas formas: a primeira, através da opção **Resource** do menu de linha do ambiente O++, onde o usuário dispõe de quatro caixas de diálogos, usadas para configurar os recursos do microcontrolador; e a segunda, através da utilização das operações específicas de configuração da porta serial, dos contadores/temporizadores, do sistema de interrupção e das portas paralelas.

A partir dos dados introduzidos nesses diálogos, o compilador gera o código necessário para programar os recursos selecionados. Se o usuário configurar os recursos através do menu de linha, a programação é feita de forma estática, ou seja, ela é feita em tempo de compilação. Caso se deseje alterar estes parâmetros em tempo de execução é preciso que o programador utilize as operações específicas de configuração, disponíveis na linguagem.

- Configuração do sistema de interrupção

No caso específico da família MCS-51, os serviços de interrupção consistem de cinco fontes de requisição de interrupção mascarada, com dois níveis de prioridade. Das cinco fontes de requisição, duas são externas, uma é para a interface serial e as últimas duas são para os Temporizadores/Contadores. Cada interrupção é atendida por uma rotina

específica de tratamento de interrupção, cujo endereço inicial para cada fonte é mostrado na tabela 5.1. A configuração das interrupções é feita através de quatro registros da CPU: TCON, TMOD, IE e IP.

As interrupções externas podem ser ativadas por nível ou por transição, dependendo da programação feita no registro TCON.

INTERRUPÇÃO FONTE	ENDEREÇO INICIAL
Requisição externa 0	0003H
Temporizador/Contador 0	000BH
Requisição externa 1	0013H
Temporizador/Contador 1	001BH
Interface serial	0023H

Tabela 5.1: Endereço inicial das rotinas de interrupção do MCS-51

Cada pedido de interrupção ativa seu correspondente flag nos registros TCON e SCON. O pedido só será reconhecido se sua máscara, presente no registro IE, estiver habilitada.

Além do registro IE, o MCS-51 dispõe de outro registro para controle das interrupções. É o registro IP, no qual se estabelece os níveis de prioridade das interrupções. Este registro é composto por 5 *bits*, cada um associado a uma fonte de interrupção diferente. Quando um destes *bits* vai a zero, indica que a interrupção correspondente tem prioridade baixa, caso contrário, a interrupção tem prioridade alta. É importante salientar que uma solicitação de interrupção não pode ser aceita durante o tratamento de uma interrupção de mesma ou maior prioridade.

O diálogo de programação das interrupções, mostrado na figura 5.6, solicita para cada fonte de interrupção os parâmetros necessários à configuração: tipo de ativação (só para as externas), a prioridade e se é para habilitar ou não àquela interrupção.

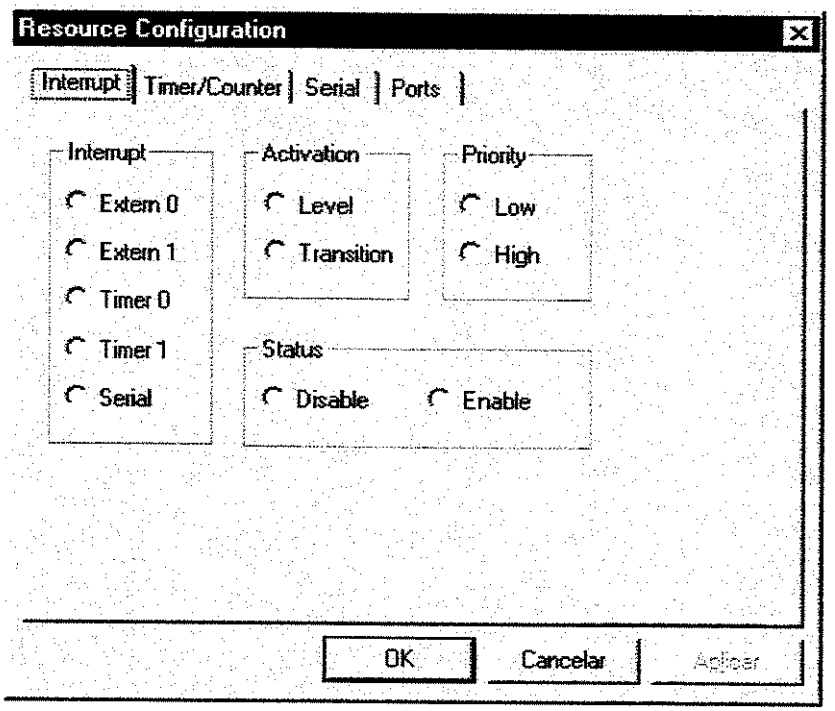

Figura 5.6: Caixa de diálogo de configuração das interrupções

• Configuração dos temporizadores/contadores

A arquitetura do MCS-51 possui dois Temporizadores/Contadores (T/Cs) de 16 *bits*, usados para medir intervalos de tempo, medir largura de pulsos, contar eventos e gerar base precisa de tempo. Os T/Cs são independentes e programados por *software*, através de alguns *bits* dos registros TCON e TMOD.

Existem quatro modos de trabalho para o T/C 0 e três modos para o T/C 1, os quais podem ser, sucintamente, descritos como mostra a tabela 5.2.

Na configuração dos T/Cs (figura 5.7) são solicitados, para cada um deles, os seguintes dados: o tipo de operação (temporização ou contagem), o modo de trabalho, a forma de ativação do T/C (campo GATE), o valor inicial de contagem, o valor de recarga (quando for o caso) e, finalmente, indicar se o T/C deve ser ligado ou não.

MODOS DE TRABALHO	DESCRIÇÃO RESUMIDA
MODO 0	Temporizador ou contador de 8 *bits* com fator de divisão de freqüência de 5 *bits* (total de 13 *bits* de contagem)
MODO 1	Temporizador ou contador de 16 *bits*
MODO 2	Temporizador ou contador de 8 *bits* com recarga automática
MODO 3	Um temporizador de 8 *bits* e um contador de 8 *bits*. Somente para o T/C 0.

Tabela 5.2: Modos de trabalho dos T/C do MCS-51

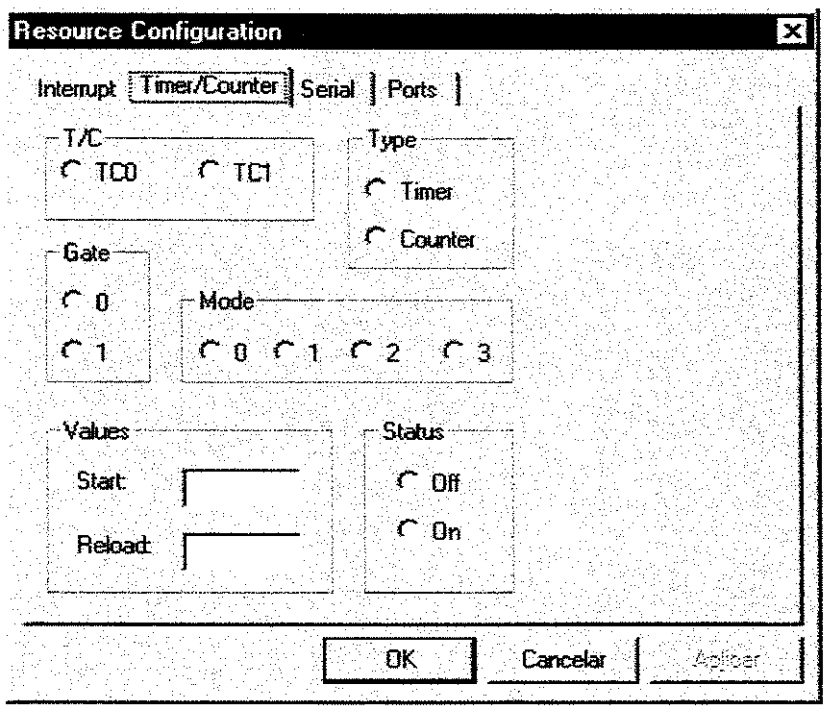

Figura 5.7: Caixa de diálogo de configuração dos temporizadores/contadores

• Configuração da interface de comunicação serial

O MCS-51 dispõe de uma porta de comunicação serial *full-duplex* que pode trabalhar tanto no modo síncrono como no modo assíncrono. Esta porta serial pode ser usada para interligar o microcontrolador a diversos dispositivos, como por exemplo: impressoras e plotters; terminais de comunicação; e computadores de maior porte. Também é possível a conexão de vários microcontroladores através da porta serial, compondo um sistema distribuído.

Na interface serial existem dois registros mapeados num mesmo endereço de memória (SBUF) usados para armazenar o dado recebido e o que será transmitido: um de leitura, que guarda o dado recebido e outro de escrita, que guarda o dado a ser transmitido. A recepção ainda dispõe de duplo *buffer*, evitando assim, o problema de sobreposição de dados recebidos.

Existe outro registro associado à interface serial, é o SCON. Ele é usado para selecionar o modo de trabalho e os parâmetros de configuração da interface e monitorar o estado de operação da mesma.

A programação dos parâmetros da interface serial é realizada pelo diálogo mostradao na figura 5.8, onde são especificados: o tamanho da palavra de dados (**Len**); a velocidade de comunicação (**Baud rate**); o tipo de paridade (**Parity**); e por último, se a interface deve ser habilitada ou não (**Enable**).

• Configuração das portas paralelas de E/S

Os microcontroladores da família MCS-51 possuem 32 pinos de E/S, agrupados em 4 portas de 8 *bits* cada. Todos os pinos podem ser individualmente configurados como entrada ou saída, sendo possível também a reconfiguração dinâmica sob o controle do *software*.

O diálogo de programação das portas de E/S, mostrado na figura 5.9, solicita para cada porta, a indicação do sentido do fluxo de dados em seus pinos, ou seja, o programador indica quais pinos vão trabalhar como entrada e quais vão trabalhar como saída.

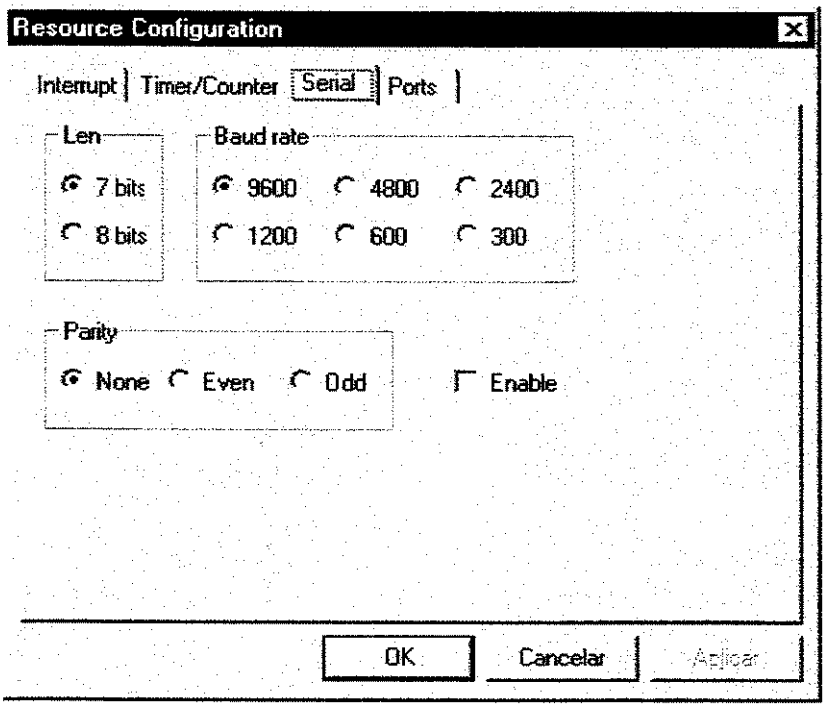

Figura 5.8: Caixa de diálogo de configuração da interface serial

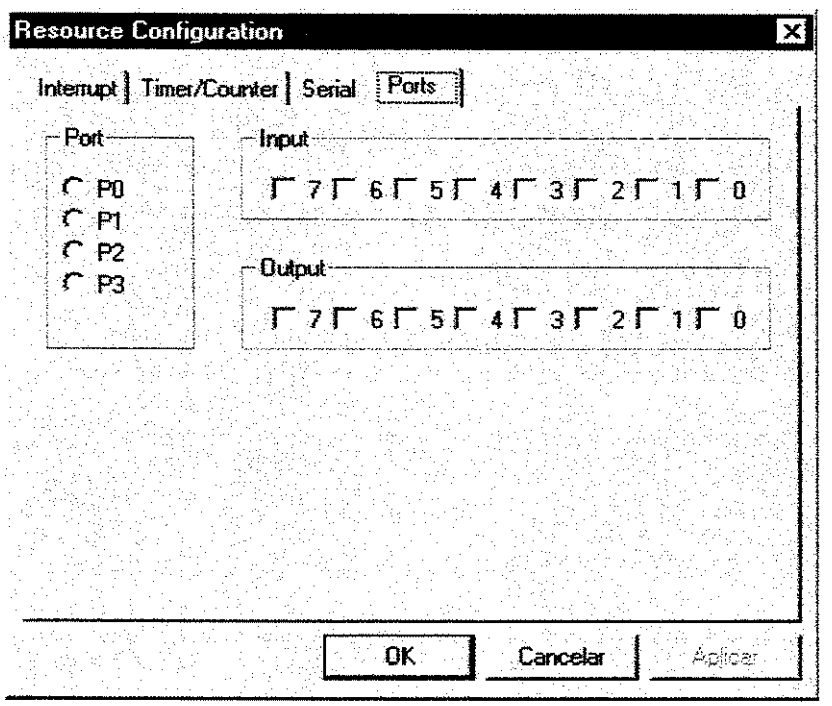

Figura 5.9: Caixa de diálogo de configuração das portas de E/S

Para melhorar o discernimento dos identificadores diretamente envolvidos nas operações de entrada e saída, foi criado um tipo especial de identificador: as portas de E/S definidas pelo usuário. Isto se justifica pelo fato da maioria das aplicações que usam microcontroladores terem uma forte interação com dispositivos de E/S, assim nada melhor que discriminar esses identificadores dos demais usados no programa, aumentando então, a clareza do programa e facilitando sua elaboração.

O diálogo relacionado às portas de E/S definidas pelo usuário, como mostra a figura 5.10, possui um campo para especificar se a porta vai trabalhar como entrada ou saída (**Type**); um campo para indicar o endereço associado (**Address**); um campo para estabelecer os bits utilizados (**Bits**); e um campo especial (**Image**) usado para inserir uma imagem que será associada àquela porta.

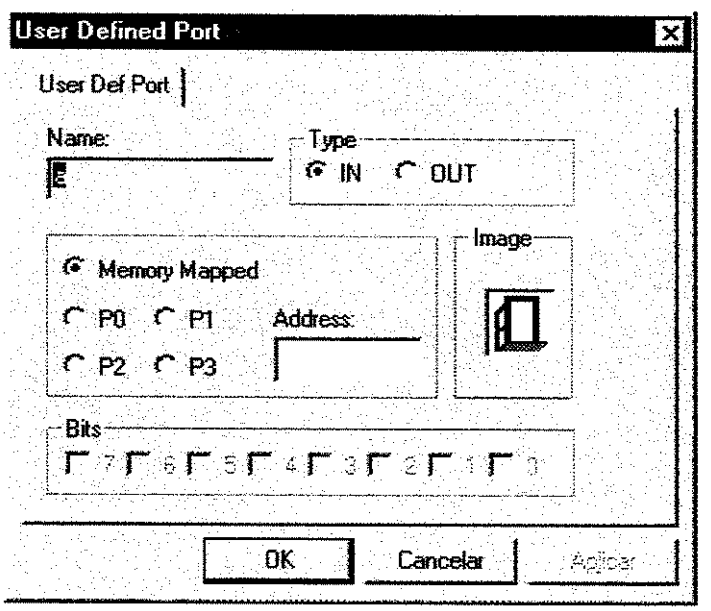

Figura 5.10: Configuração das portas de E/S definidas pelo usuário

5.1.2. Um programa exemplo

Este item aborda como se realiza a implementação de uma aplicação usando a linguagem O++. Considere uma aplicação hipotética, mas muito ilustrativa, que trabalha com aquisição de dados. Trata-se de uma aplicação que deve ler 100 amostras de dados analógicos (através de um conversor analógico-digital de 8 bits) e enviar por uma porta serial o valor médio dessas amostras. Para isso será definida uma classe chamada *ADC* que modela o conversor analógico-digital real. Esta classe (ver figura 5.11) é composta por um valor para armazenar a última amostra convertida (*value*), um método de aquisição da amostra (*Aquire*) e um método para realizar a manipulação do conversor (*Convert*).

O método *Convert*, mostrado também na figura 5.11, envia um sinal de início de conversão através da porta de saída chamada *START* e espera que a conversão se conclua, monitorando o sinal da porta de entrada chamada *EOC* (End Of Conversion). Quando o conversor tiver uma amostra disponível, ela é lida pela porta de entrada *DATA* e armazenada no membro *value*. Todas essas portas devem estar previamente definidas, para isso se utiliza o diálogo de descrição de portas de e/s, mostrado anteriormente no item 5.1.2.3.

Toda aplicação inicia sua execução no módulo de partida chamado *main*. No caso desse exemplo, o módulo *main* (ver figura 5.12) cria os seguintes elementos: o objeto chamado *adc808* da classe *ADC*; uma tabela para armazenar as amostras (*tab*); um elemento para armazenar o valor médio das amostras (*mean*); além de elementos auxiliares usados para contagem (*cont*) e retenção de amostra (*data*).

O módulo *main* inicia sua execução com a configuração da porta serial, cujos parâmetros são introduzidos via diálogo mostrado anteriormente no item 5.1.2.3. Depois há um laço no qual estão as operações de envio de mensagem para o conversor adquirir uma amostra e o armazenamento dessa amostra na tabela. Posteriormente vem a operação de cálculo do valor médio e, finalmente, o envio desse valor na porta serial.

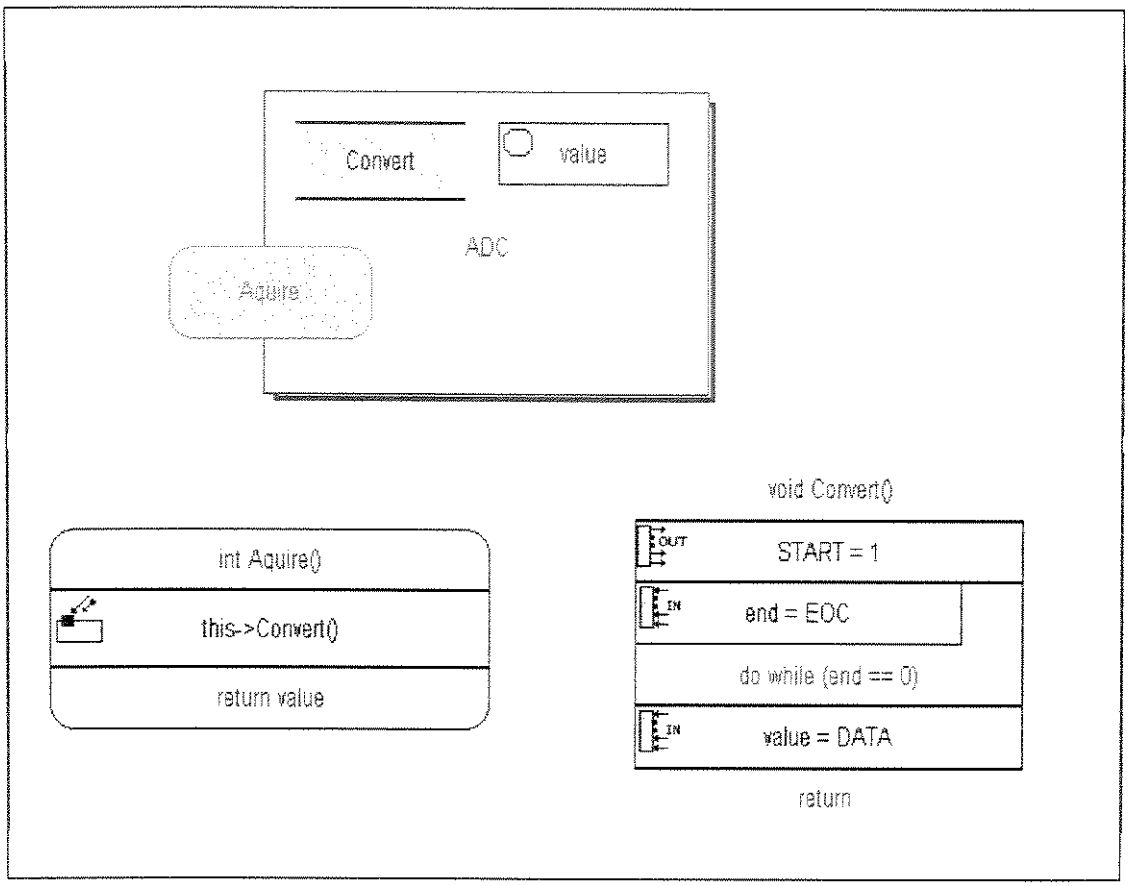

Figura 5.11: A classe *ADC*

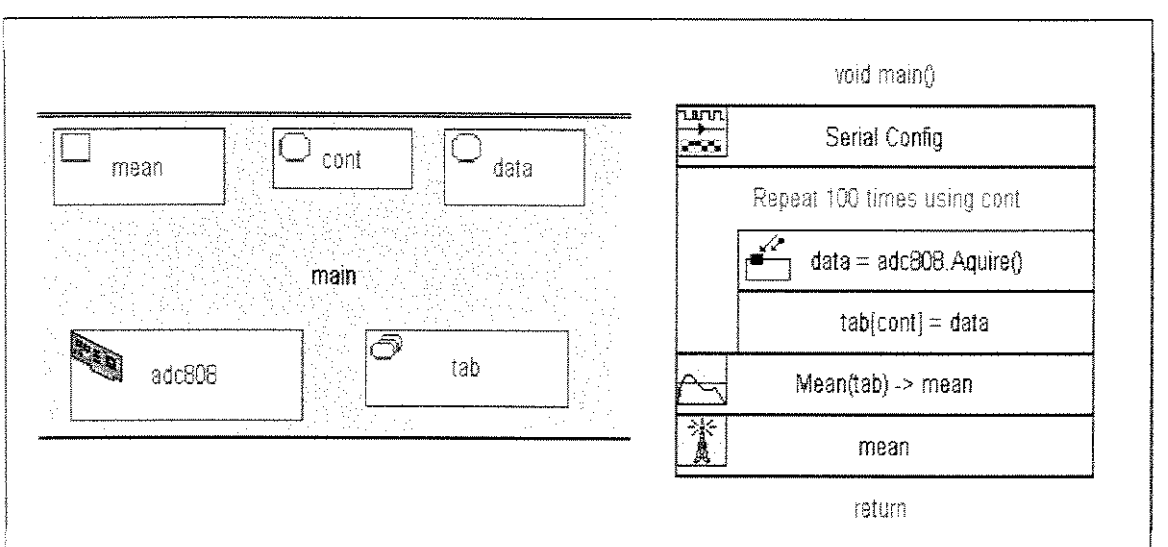

Figura 5.12: O módulo de partida *main*

Para ilustrar como se consegue realizar mudanças de forma bastante produtiva, a aplicação anterior será modificada para trabalhar com um conversor de 16 *bits*. Neste caso cria-se uma classe chamada *ADC16* derivada da classe *ADC*. Esta nova classe, mostrada na figura 5.13, herda os métodos e atributos da classe pai e faz uma implementação específica para o método de conversão, pois o dado convertido é de 16 *bits*. Assim, esse método obtêm os valores da parte menos significativa do valor convertido (através do método *Convert* da classe pai) e da parte mais significativa (através da leitura da porta *DATAHI*, que foi previamente definida). Logo em seguida, é feita a junção desses valores e levado o valor final para o membro *value*. Dessa forma, o módulo *main* pode mudar a classe do objeto *adc808* e trabalhar com 16 bits, sem ter que alterar suas operações.

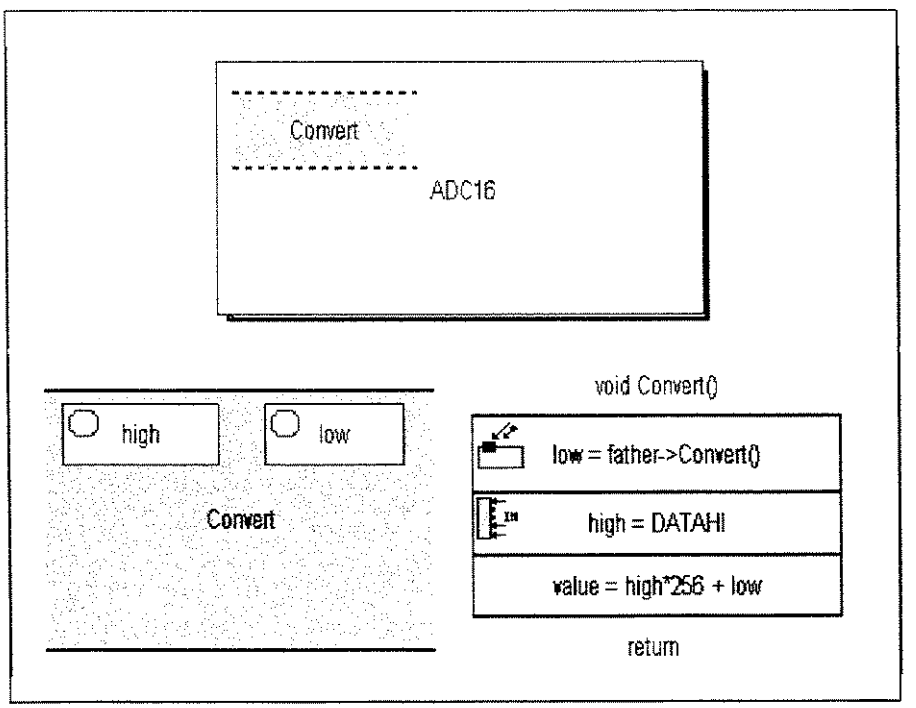

Figura 5.13: A classe *ADC16*

5.2. A implementação do compilador

5.2.1. A escolha do ambiente de desenvolvimento da linguagem O++

Como um dos objetivos principais do trabalho era criar uma linguagem visual para programação de microcontroladores, optou-se por usar um ambiente de desenvolvimento que explorasse os recursos gráficos na implementação da interface homem-máquina. Assim, o ambiente escolhido foi o Windows da Microsoft, que é atualmente o sistema operacional mais difundido para microcomputadores.

O Microsoft Windows surgido em meados da década de 80, apresenta uma interface com o usuário baseada no "ver e sentir" das normas CUA (Common User Access) da IBM. Faz forte valorização ao visual (gráfico, mouse e janelas) na comunicação homem-máquina e recebeu grande influência dos ambientes orientados a objetos. Ele é dirigido a eventos e permite o reaproveitamento de objetos de classes pré-definidas, que são usadas em programação baseada em hereditariedade.

A escolha da linguagem de programação se deu pelas facilidades oferecidas pela metodologia orientada a objetos e pelos recursos de manipulação de estruturas gráficas. Desta forma, optou-se pelo Visual C++ por apresentar essas características e ser um pacote totalmente compatível com o Windows, pois foi desenvolvido pela própria Microsoft.

O Visual C++ é um programa de desenvolvimento de aplicações orientadas a objetos. Além de um Compilador, de um Linkeditor e de um Depurador, ele integra num mesmo pacote outras ferramentas de desenvolvimento, desde editor de programa até um gerador automático de código para aplicações básicas. Tais ferramentas formam um ambiente interativo de desenvolvimento baseado no Windows, chamado Visual Workbench. [KRU96]

Uma das ferramentas mais interessantes do Visual C++ é, sem dúvidas, o editor de recursos, o App Studio. Este editor é responsável pela criação de *menus*, diálogos, *bitmaps*, ícones, tabelas de *string* e cursores de forma bastante amigável com o programador. Estes componentes são compilados para um arquivo especial em disco (o arquivo de recursos) ou vinculados ao programa executável. [KRU96]

Para facilitar o processo de codificação o Visual C++ dispõe de um gerador automático de código, o App Wizard, que cria a estrutura básica de uma aplicação para

Windows com recursos, classes e arquivos definidos através de caixas de diálogos, introduzindo de forma rápida o programador na nova aplicação. Com o App Wizard pode-se elaborar o esqueleto de uma aplicação genérica e depois acrescentar as implementações específicas. [KRU96]

Com a finalidade de tornar o processo de manutenção menos tedioso, o Visual C++ oferece uma ferramenta de inspeção chamada Source Brouser que permite examinar uma aplicação a partir de uma classe ou de uma função. [KRU96]

O Visual C++ possui também um programa gerenciador de classes, o Class Wizard, que proporciona a criação de protótipos, funções e o código para ligar as mensagens à estrutura da aplicação. [KRU96]

Finalmente pode-se destacar no Visual C++ a Microsoft Fundation Class ou apenas MFC. A MFC é uma biblioteca de classes criada para facilitar o desenvolvimento de aplicações orientadas a objetos. Ela possui cerca de cem classes agrupadas em três categorias: a) classes destinadas à construção de aplicações Windows, que oferecem fácil acesso aos dispositivos de interface gráfica e permitem a incorporação e "linkagem" de objetos; b) classes de propósito geral, usadas para manipulação de listas encadeadas, arquivos, *strings*, dentre outras; c) Macros e Globais, que consistem em várias macros e objetos de propósito geral. [KRU96]

5.2.2. Arquitetura do ambiente O++

Um programa em O++ é constituído tanto por estruturas gráficas (declaração de classes, declaração de dados e operações representadas por cartas NS) como por estruturas textuais (operações lógicas e aritmética, referência à elementos do programa, dentre outras). Estas estruturas são introduzidas no ambiente de programação (figura 5.14) através de um módulo chamado *Editor Híbrido*, que é composto por um editor gráfico específico para o desenho das estruturas gráficas definidas na linguagem; e um editor de texto para um subconjunto do C++.

Esses dois sub-módulos trabalham em conjunto, gerando uma estrutura que agrega a informação visual com a complementar informação textual. Esta estrutura, por sua vez, é passada ao módulo de filtragem (*Filtro*), que separa as declarações gráficas das declarações

em C++. As declarações gráficas são conduzidas ao *Compilador Gráfico* e as declarações em C++ são passadas ao módulo *Joiner*.

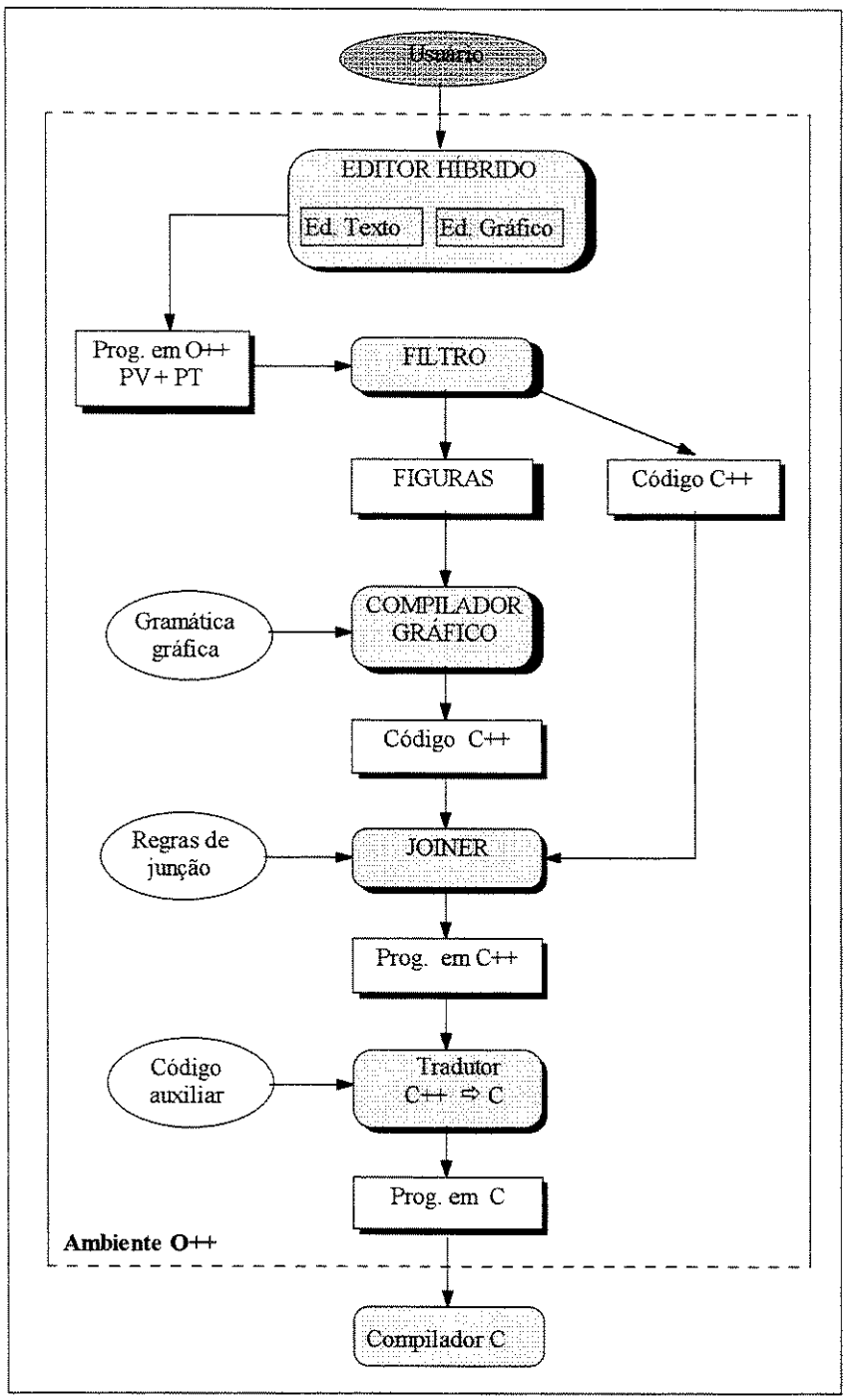

Figura 5.14: Diagrama em blocos do compilador O++

O Compilador Gráfico, através de uma gramática gráfica, gera possíveis mensagens de erro e traduz as declarações gráficas para o respectivo código em C++. Tal código é, então, complementado com as declarações em C++, que foram inseridas diretamente pelo programador, formando o programa final em C++. Este último passo é executado pelo módulo Joiner, que através de regras de junção, estabelece a ligação entre os dois códigos.

Finalmente, há um *Tradutor* (C++ ⇨ C) utilizado para converter o código gerado em C++ para a linguagem C. Para que isto seja possível, é necessário introduzir no código final, um conjunto de rotinas e estruturas de dados que permitam implementar as estruturas orientadas a objetos do programa em C++.

A necessidade da geração de código em linguagem C é devido a não disponibilidade comercial de um compilador C++ para vários tipos de microcontroladores, como o MCS-51 e o 68HC11.

5.2.3. Modelamento da linguagem O++

Para entendermos como a linguagem O++ foi implementada, temos inicialmente que analisar o modelo formal de uma linguagem visual. O modelo escolhido foi o modelo proposto em [BOT95], que consiste em:

- Qualquer figura pode ser representada por uma imagem digital.

A **imagem digital** *i* é um conjunto bidimensional

$$i : \{1,...,l\} \times \{1,...,c\} \rightarrow P$$

onde: *l* é o número de linhas da imagem;
c é o numero de colunas da imagem; e
P é o alfabeto de pixel.

- Uma **linguagem pictorial** LP é um subconjunto do conjunto I das possíveis imagens digitais.

$$LP \subseteq I$$

- A **descrição** *d* é um conjunto de fatos de alguma **linguagem de descrição** LD.

- Uma **interpretação** *int* de uma imagem *i* é a função que leva à sua descrição:

int : LP $\rightarrow$ LD.

Obs.: Uma interpretação é especificada por um observador da figura, e é baseada no **significado** dela.

- A **materialização** *mat* de uma descrição *d* é uma função que leva à sua visualização:

mat : LD $\rightarrow$ LP.

Obs.: A materialização é a representação pictorial de uma descrição.

- Uma **sentença visual** é a tripla

$(i, d, (int, mat))$.

Obs.: Uma sentença visual é chamada de:

- **fiel** sse $i = mat(d)$.
- **cheia** sse $d = int(i)$.
- **completa** sse ela é fiel e cheia.

- Uma **linguagem visual** é um conjunto de sentenças visuais.

A escolha desse modelo se deve por ele ser um modelo genérico, mas que possibilita que uma interpretação seja especificada por mais de um observador. Permite-se, assim, trabalhar com mais de um significado para cada figura. É importante salientar que esses significados dizem respeito ao tipo de operação que o programador deseja realizar com cada imagem. Num dado momento, por exemplo, o programador pode solicitar ao ambiente de programação que armazene em disco as informações de um elemento do programa (expresso por uma imagem); em outro momento, ele pode solicitar que seja gerado o código daquele elemento.

Pelo exposto, pode-se perceber que cada imagem possui mais de uma descrição, cada uma dentro de um contexto. Este detalhe foi facilmente implementado, pois usamos uma linguagem orientada a objetos, o Visual C++. Cada elemento visual de um programa em O++ é implementado como um objeto em C++, de tal forma que o elemento possui uma imagem e um conjunto de descrições (métodos associados àquele elemento). Assim, dependendo do contexto que seja realizada uma interpretação, o respectivo método é chamado, fazendo com que seja gerada a descrição pertinente.

5.2.4. A estrutura do código e os mecanismos de tradução

Neste ponto, vamos analisar a descrição dos elementos visuais de um programa em O++ no contexto da compilação. Usamos o modelo cliente/servidor para implementar a arquitetura do compilador. Nela, cada elemento visual do programa, implementado como um objeto, possui um método de tradução, chamada de **tradutor servidor**. Este método é responsável pela tradução das declarações gráficas para o código C++, pela junção das declarações em C++ introduzidas diretamente na figura e pela tradução final para a linguagem C.

Há também um objeto, chamado **tradutor cliente**, que é ativado através da interface com o programador. Este objeto tem a função de desencadear a seqüência de chamadas às mensagens de tradução dos elementos do programa. Além disso, ele realiza a concatenação final de todos os fragmentos de código (veja figura 5.15).

A linguagem O++ possui um código cuja estrutura é orientada a figuras (elementos visuais do programa), ou seja, as declarações e operações são descritas por arranjos de símbolos padronizados. Estes arranjos possuem uma sintaxe dirigida pelo editor gráfico, de tal forma que só são permitidas combinações corretas de declarações e operações. Isto além de diminuir excessivos testes de sintaxe, como ocorre nas linguagens tradicionais, torna a estrutura do código mais previsível para o compilador sem, no entanto, obrigar o programador a seguir rigidamente uma sintaxe como no caso das linguagens de formato fixo.

Cada elemento visual do programa é responsável pela geração de seu próprio código ou da maior parte dele. Isto facilita o processo de tradução e diminui a necessidade de se usar otimizadores de código. Cada figura se comunica com os outros elementos envolvidos na declaração ou operação para que eles gerem a parte do código a eles pertinente ou indiquem a natureza dos parâmetros a serem manipulados. Assim, cada figura pode determinar o melhor código possível, criando-o semanticamente correto. Na maioria dos casos, por exemplo, cada figura não compõe o código de acesso aos parâmetros que ela venha a utilizar, isso fica a cargo dos objetos que implementam tais parâmetros. Cada uma apenas envia mensagens a esses objetos para que eles gerem o código de acesso aos parâmetros utilizados pela figura, que também devem ser elementos visuais.

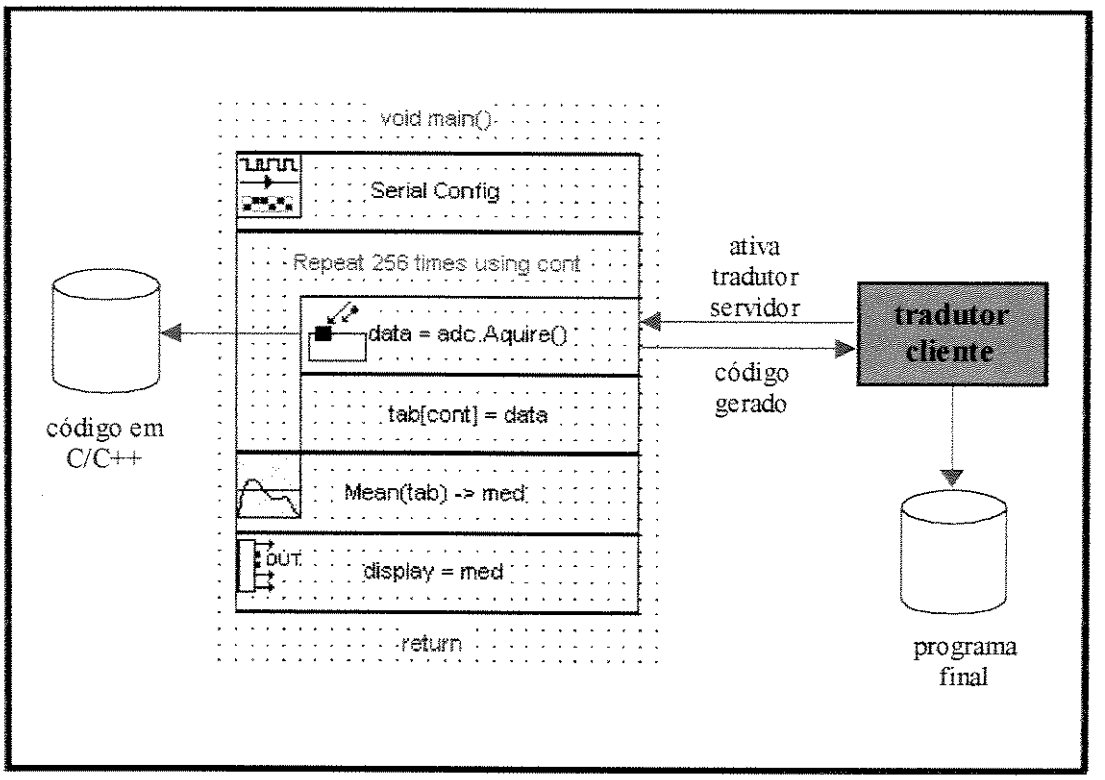

Figura 5.15: Arquitetura de tradução

Evidentemente, nem todas as redundâncias são eliminadas. Em operações que envolvem mais de uma vez um determinado elemento, pode ocorrer geração de código redundante, pois para cada novo acesso ao elemento é enviada uma mensagem ao objeto que o representa, a fim de que este gere o código de acesso àquele elemento.

Na geração do código, procurou-se adotar um mecanismo que fosse o mais genérico possível, no que diz respeito ao tipo de microcontrolador. Por isso a maior parte do código gerado utiliza, fundamentalmente, as declarações essenciais do ANSI C. Obviamente, nas operações associadas às características do microcontrolador isto não é possível, pois cada microcontrolador possui suas particularidades.

Entre os compiladores C para microcontroladores pode haver pequenas diferenças sintáticas. Desta forma, adotou-se no processo de tradução uma **tabela de códigos fonte** que armazena um conjunto de declarações em C/C++ necessárias para implementar cada declaração ou operação da linguagem O++. Esta tabela, mostrada no anexo III, é guardada em disco num arquivo de formato ASCII.

Para cada compilador C utilizado é necessário um arquivo que contenha a tabela de códigos fonte correspondente. A linguagem O++ não dispõe de nenhuma ferramenta de apoio para edição desta tabela, entretanto ela pode ser modificada através de um editor de textos convencional.

5.2.5. Geração de código em linguagem C

Apesar das vantagens oferecidas pelas linguagens orientadas a objetos, a linguagem C continua sendo a mais conhecida e mais difundida dentre as linguagens de programação de sistemas microcontrolados. A maioria dos fabricantes de microcontroladores não oferece outra alternativa além das linguagens C e Assembly. Assim sendo, os conceitos da programação orientada a objetos, disponíveis na linguagem O++, foram traduzidos para a linguagem C, e como ela não oferece suporte à orientação a objetos, houve a necessidade de introdução de código auxiliar, criando estruturas de dados para implementar as classes e os mecanismos de comunicação entre objetos.

Conforme comentado no capítulo 1, os métodos de implementação dos conceitos da programação orientada a objetos, usando a linguagem C para microcontroladores, propostos em [SAM97] e [SIC97] conduzem a geração excessiva de código e a necessidade de mais memória.

A maioria dos conceitos da programação orientada a objetos são representados visualmente na linguagem O++, que dispõe de um ambiente de programação altamente amigável e direcionado para aplicações projetadas usando esse tipo de paradigma. Além disso, como ela possui uma sintaxe dirigida por um editor gráfico, só são permitidas combinações corretas de declaração. Desta forma, a componente humana exerce muito pouco influência na geração do código orientado a objetos em C, pois o programador não trabalha diretamente com esse código. Ele descreve as classes e os algoritmos de processamento de mensagens através de uma sintaxe (visual) própria da linguagem O++. Esta sintaxe, ao contrário do que ocorre nos dois métodos já comentados, não apresenta subterfúgios para o tratamento dos conceitos da programação orientada a objetos, ao contrário, ela foi projetada para tornar estes conceitos mais intuitivos e mais acessíveis aos programadores.

Ao criar uma sintaxe artificial de descrição de classes e comunicação entre objetos, os dois métodos mencionados, impõem regras ao programador. Essas regras são propositais e necessárias, dentro do contexto dos dois métodos. Por um lado é preciso fazer o programador criar código orientado a objetos, por outro, o formato deste código deve ser apropriado com os algoritmos que converterão tal código em código procedimental C.

Na linguagem O++ essas regras praticamente não existem. A única disciplina, com a qual o programador se detém, é a imposta pelo próprio estilo da programação orientada a objetos. Por exemplo, para determinar se um membro é público ou protegido, o programador apenas arrasta o membro para fora ou dentro, respectivamente, da área de descrição da classe. Assim, ao invés de impor restrições, com o objetivo de afunilar as possibilidades de tratamento das estruturas orientadas a objetos por parte do programador, a linguagem O++ faz uso de mecanismos didáticos para tratar estas estruturas.

Pelo exposto, pode se perceber que o código orientado a objetos em C, gerado automaticamente pelo compilador O++, é mais previsível do que os gerados pelas versões manuais das duas propostas anteriores e, portanto, as estruturas de dados e o código adicional de iniciação, gerados pelo O++, podem ser melhor otimizados.

Para melhor ilustrar o processo de geração de código em linguagem C, vamos examinar um pequeno exemplo: trata-se da implementação de uma classe para contadores, a classe ***counter*** mostrada na figura 5.16. Esta classe possui um atributo protegido usado para armazenar o valor de contagem (*value*); os métodos públicos *get* e *set* para ler e escrever o valor de contagem, respectivamente; e, finalmente, um método, também público, para ativar o incremento, que no caso é sempre de um.

O compilador O++, ao gerar o código em C (veja listagem 5.1), define cada classe como sendo um novo tipo de dado (linhas 1-4). Especificamente, uma classe é definida como sendo uma estrutura composta pelos seus membros de dados. Os métodos são definidos fora desta estrutura (linhas 6-19), mas ao nome de cada um deles é acrescido um identificador da classe (o nome da classe). Apenas os métodos virtuais, aqueles usados para implementar o polimorfismo, possuem um apontador dentro da estrutura de definição da classe. Com isso se economiza memória, pois é feita uma diminuição do número de apontadores para métodos.

Os métodos de uma classe são compartilhados pelos objetos desta classe ou de classes derivadas a ela, assim, para que os métodos possam manipular adequadamente os membros de dados dos objetos, foi introduzido um parâmetro extra para armazenar o apontador para o objeto corrente (linhas 6, 11 e 16).

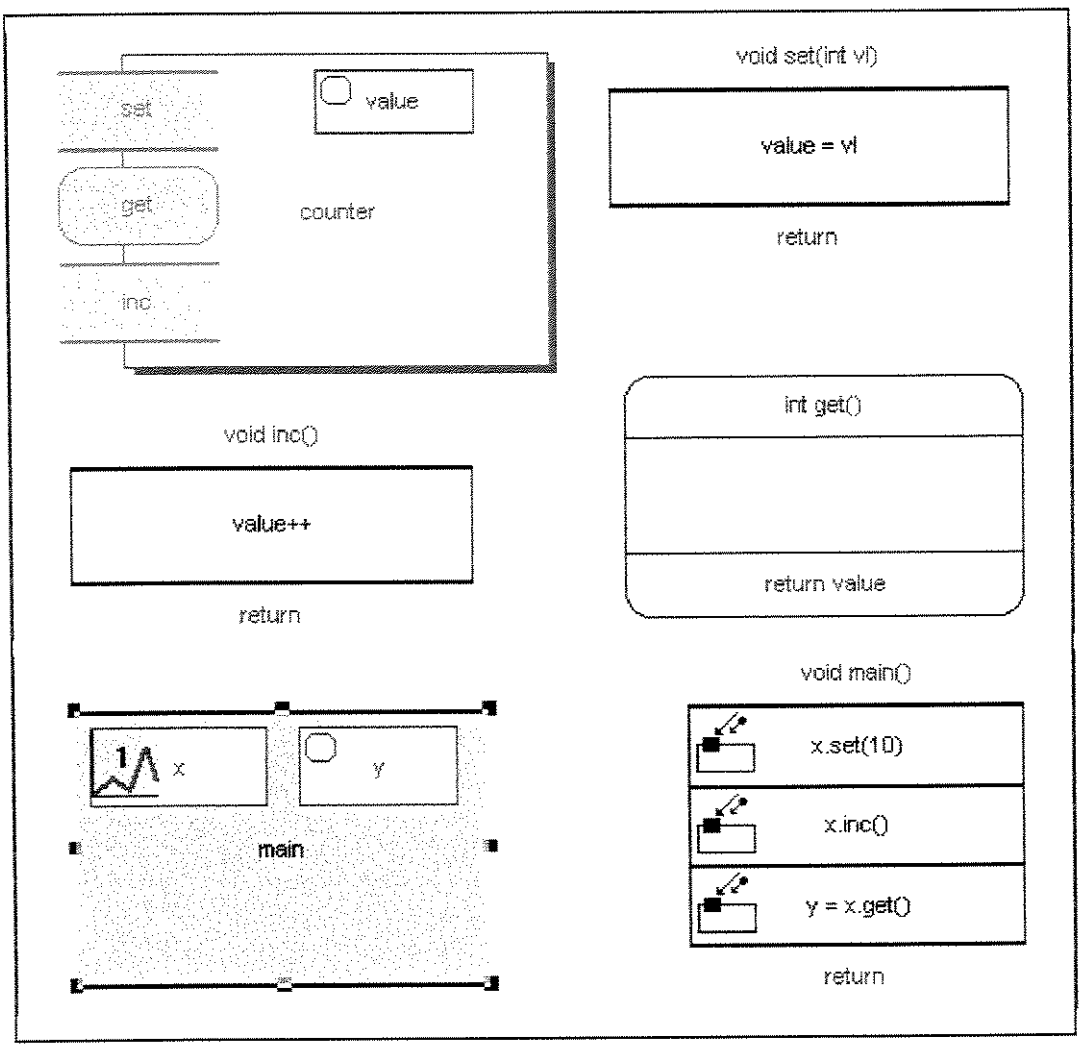

Figura 5.16: A classe *counter* implementada em O++

A criação dos objetos segue a mesma forma da criação de variáveis. Na linha 23, do mesmo exemplo, temos a criação do objeto *x*, que pertence à classe definida *counter*. Neste caso não foi ativado o construtor da classe (ele nem se quer foi definido), uma vez que não houve necessidade de nenhuma iniciação. Porém, o programador pode definir um construtor para cada classe, e quando um objeto for criado, o construtor da classe deste objeto é automaticamente ativado.

O envio de mensagens a um objeto (linhas 26-28) se processa pela chamada ao método responsável pelo tratamento da mensagem. Nesta chamada sempre se passa o apontador para o objeto que vai receber a mensagem, como já comentado anteriormente.

```
1       #define _counter
2               int value;\
3
4       typedef struct { _counter } counter;
5
6       void _counter_set (counter* this, int vl)
7       {
8               this→value = vl;
9       }
10
11      void _counter_inc (counter* this)
12      {
13              this→value++;
14      }
15
16      int _counter_get (counter* this)
17      {
18              return (this→value);
19      }
20
21      void main(void)
22      {
23              counter x;
24              int y;
25
26              _counter_set ((counter*)&x,10);
27              _counter_inc ((counter*)&x);
28              y = _counter_get ((counter*)&x);
29      }
```

Listagem 5.1: A classe *counter* implementada em C

5.2.6. Considerações sobre encapsulamento e ocultação de informação

Pelo que foi apresentado, pode-se perceber que a versão em C não garante o encapsulamento de membros de dados e métodos dentro de uma única unidade, pois os métodos são definidos fora da estrutura que comporta os membros de dados. Também pode-se observar que a versão em C não assegura a ocultação de informação, uma vez que tanto os membros públicos quanto os membros protegidos podem ser manipulados fora do escopo da classe ou de suas descendentes.

Essa análise seria válida, se pensássemos que o programador iria construir manualmente o código em C, entretanto isto não ocorre. O código em C é gerado pelo compilador O++, que possui um ambiente próprio para garantir que os conceitos da programação orientada a objetos sejam, adequadamente, utilizados pelo programador.

Para entender como o ambiente O++ garante os conceitos de encapsulamento e ocultação de informação, vamos voltar ao exemplo anterior. Na listagem 5.1 vemos claramente que no módulo *main*, mesmo sem termos uma instância da classe *counter*, é possível ativar os métodos desta classe. Isto desconfigura o encapsulamento, porém o ambiente O++ não permite que esses métodos sejam ativados sem que seja criado um objeto da classe correspondente.

Considere a situação apresentada na figura 5.17. Toda vez que a operação de chamada a um método for selecionada, a vista de catálogo dispõe apenas os métodos relativos aos objetos criados naquele escopo, obedecendo os critérios de público e protegido. No caso da figura, são dispostos os métodos públicos da classe *counter*, pois há uma instância desta classe no módulo *main*, o objeto *x*. Já se o escopo for um método de uma classe, todos os métodos da mesma estariam disponíveis, tanto os público como os protegidos. Em outras palavras, o ambiente não permite que um atributo ou operação seja usado fora da classe (a qual ele pertence ou é herdado) ou fora do escopo de alguma instância dela ou de uma herdeira.

Vamos agora analisar como o ambiente O++ possibilita o ocultação de informação. Tomando os exemplos das figuras 5.18a e 5.18b, pode-se observar que no escopo do método *inc*, que pertence à classe *counter*, tem-se acesso ao membro protegido *value*, desta mesma classe. Entretanto, no escopo do módulo *main* o membro *value* não aparece disponível, pois o mesmo é, como já dito, protegido. Ele só pode ser manipulado pelos

métodos da classe *counter* ou de suas herdeiras. Com isso se garante ocultação tanto de membros de dados como de métodos.

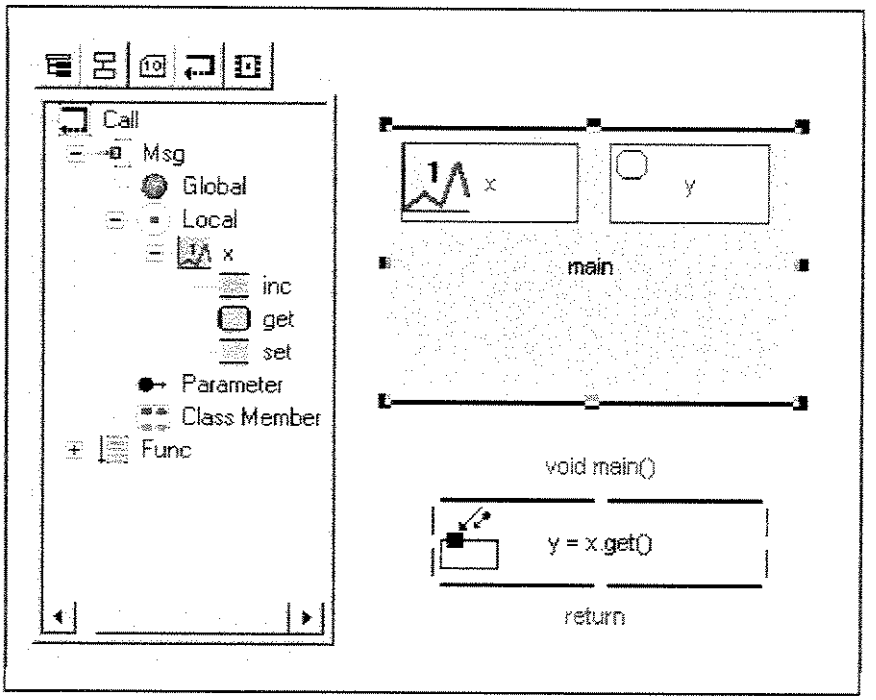

Figura 5.17: Os métodos no escopo do módulo *main*

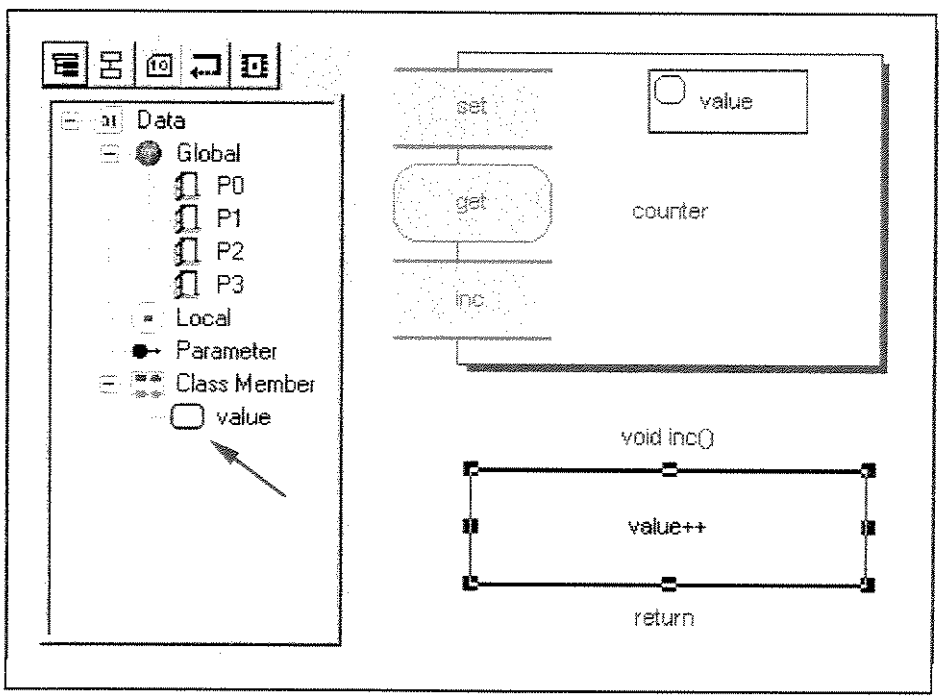

Figura 5.18a: Os atributos no escopo do módulo *counter::inc*

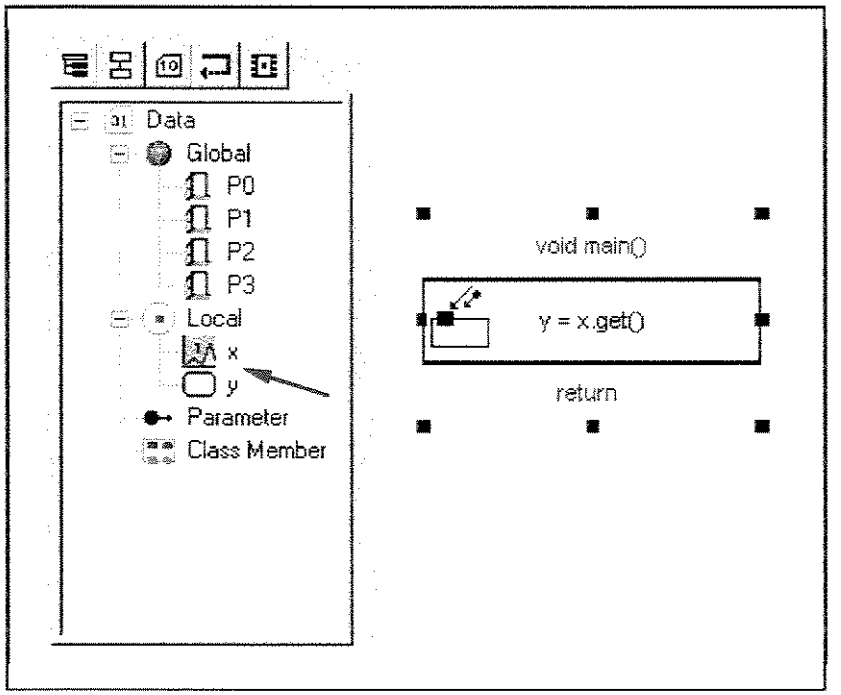

Figura 5.18b: Os atributos no escopo do módulo *main*

5.2.7. Considerações sobre hereditariedade e reusabilidade

Na linguagem O++, como em outras linguagens orientadas a objetos, o reuso de componentes de programa é obtido ao se definir novas classes a partir de outras já existentes, ou seja, uma nova classe pode herdar atributos e operações de uma classe antecessora. A classe herdeira, também chamada classe filha ou derivada, pode incluir novos atributos e operações, não contidos na classe pai. Pode, ainda, especializar atributos e operações já existentes.

O processo de implementação de herança na linguagem O++ será examinado através do exemplo a seguir. A classe *counter*, vista anteriormente, será especializada numa classe *counter2* (ver figura 5.19), que herda todas as características da classe anterior, porém permite incremento diferente de 1. Esta nova classe mudará o método de incremento, acrescentará o atributo valor de incremento (*vl_inc*), bem como criará um método (*set_inc*) para mudar este valor (ver figura 5.20).

Assim, se observarmos a listagem 5.2, podemos notar que na definição da classe *counter2* (linhas 1-4) ocorrem as inclusões dos atributos da classe *counter* e do novo atributo *vl_inc*. Nas linhas 6-14 são implementados os métodos *inc* e *set_inc*. O primeiro

sobrepõe o método de incremento da classe pai e, o segundo é um novo método da classe filha.

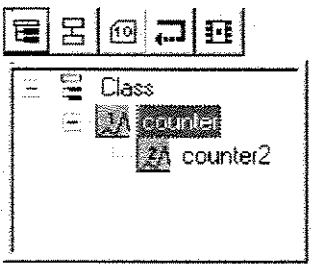

Figura 5.19: Um exemplo de hierarquia de classes

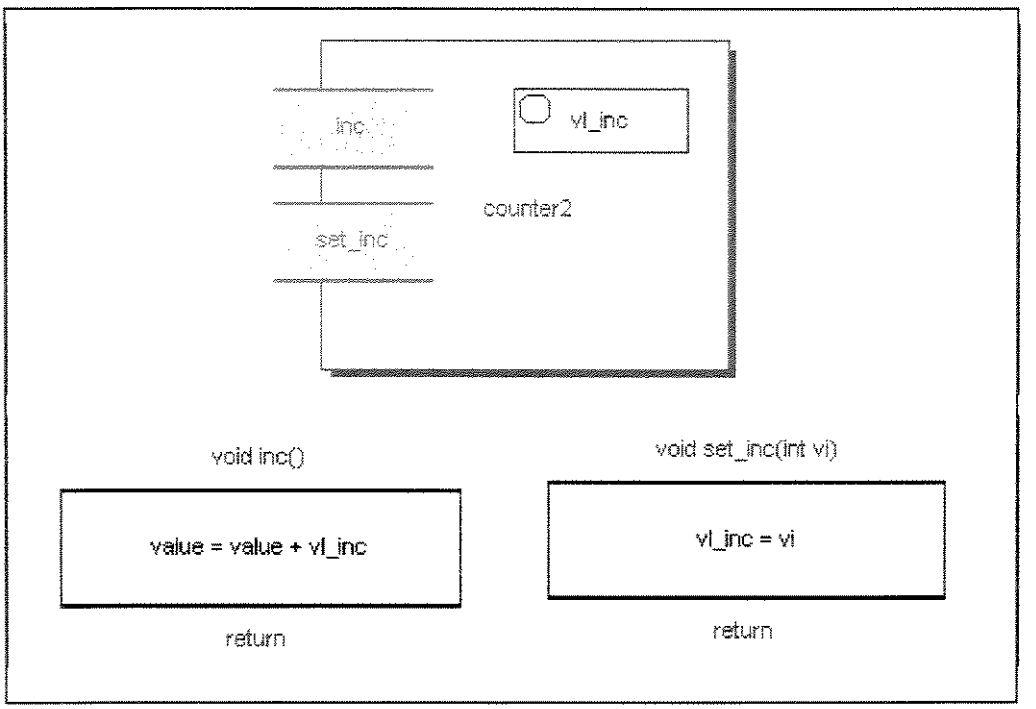

Figura 5.20: A classe *counter2* implementada em O++

Da mesma forma como ocorreu com o encapsulamento e ocultação de informação, a implementação da hereditariedade está muito ligada ao ambiente de programação.

Ao se criar uma classe, a partir de uma já existente, o ambiente O++ incorpora à nova classe, todos os atributos e operações da classe herdada. Tal processo é automático, permitindo o reuso efetivo de componentes de programa. Se voltarmos à classe *counter2* podemos ver como ocorre o reuso ao se criar um objeto desta classe no módulo *main*, por exemplo (figura 5.21). Ao se criar o objeto *z*, o ambiente torna disponível os membros

públicos *inc* e *set_inc*, que pertencem à classe *counter2*. O ambiente dispõe também os métodos públicos da classe *counter*, pois a mesma é a classe pai de *counter2*.

```
1       #define _counter 2      _counter
2                               int vl_inc;\
3
4       typedef struct { _counter2 } counter2;
5
6       void _ counter2_set_inc (counter2* this, int vi)
7       {
8               this→vl_inc = vi;
9       }
10
11      void _ counter2_inc (counter2* this)
12      {
13              this→value = this→value + this→vl_inc;
14      }
```

Listagem 5.2: A classe *counter2* implementada em C

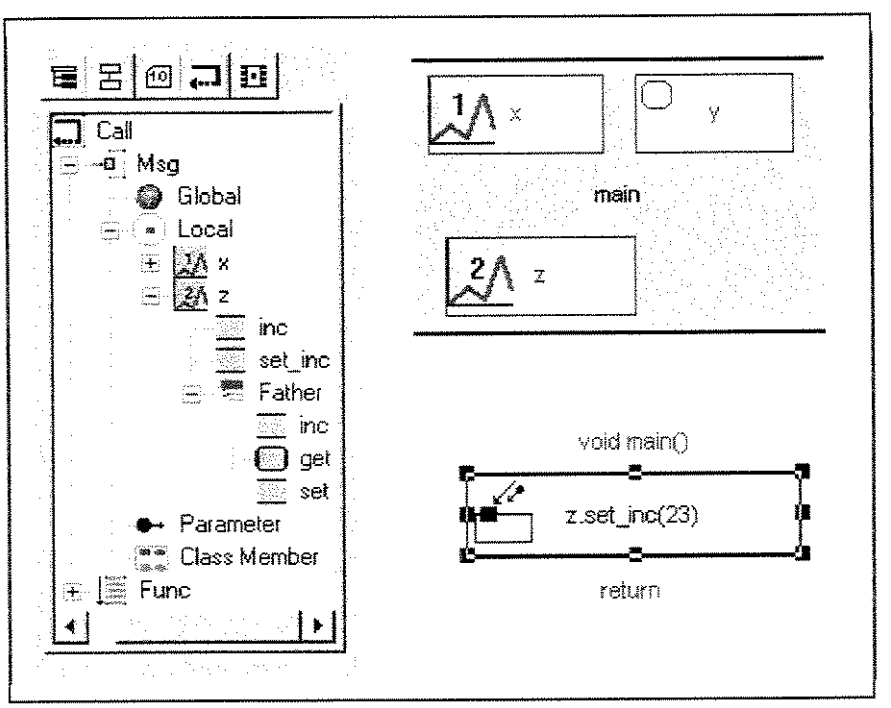

Figura 5.21: Os métodos no escopo do módulo *main*

5.2.8. Considerações sobre polimorfismo

Como se sabe, o polimorfismo nas linguagens orientadas a objetos é um fenômeno no qual a mesma mensagem pode resultar em ações completamente diferentes quando recebidas por objetos diferentes.

A linguagem O++ oferece suporte ao polimorfismo através da criação de métodos virtuais. Um método é chamado **virtual** quando cada classe, dentro de uma mesma linhagem, possui uma implementação própria para este método, sem mudar seu nome. Além da sobrecarga do nome, a adequada implementação é determinada em tempo de execução. O sistema determina o endereço do método que será executado de acordo com a classe do objeto que o ativou. Esse mecanismo é chamado **junção posterior** e requer, em cada objeto, um apontador para cada método definido como virtual.

Para melhor entendermos o mecanismo utilizado para implementar o polimorfismo na linguagem O++, vamos analisar o exemplo a seguir. Considere que tenhamos uma classe chamada *Figura* (mostrada na figura 5.22), com os membros de dados *LA* e *LB*, representando a base e a altura da figura; e o membro de dados *area*, que armazena a área da figura. Além disso, considere que essa classe possui um método para iniciar os valores *LA* e *LB*, chamado *SetLados*; um método virtual para calcular a área da figura, chamado *CalcArea* (observe que a simbologia deste método é diferenciada por linhas tracejadas); e outro para obter o valor da área calculada, chamado *GetArea*.

Da classe *Figura* são herdadas a classe *Triangulo* e a classe *Retangulo*, conforme mostra a figura 5.23. Em cada uma dessas classes o método de cálculo da área da figura é redefinido no contexto da classe, como ilustram as figuras 5.24 e 5.25.

Para completar o programa exemplo, considere o módulo *main* mostrado na figura 5.26. Este módulo possui um objeto da classe *Triangulo* e outro da classe *Retangulo*, chamados de *x* e *y*, respectivamente.

Agora é preciso entender como o método *GetArea* funciona. Este método chama o método que calcula a área da figura (*CalcArea*) e retorna o atributo *area*, ao qual é introduzido o valor da área calculada. Como cada classe possui uma implementação própria para o método *CalcArea* e o método *GetArea* possui apenas uma implementação, é necessário que cada objeto possua um apontador para a correspondente implementação do método virtual *CalcArea*. Observando o código gerado pelo compilador, que se encontra no

anexo IV, podemos ver que o apontador é definido na classe base *Figura* (linha 9), sendo herdado pelas classes filhas. A iniciação desse apontador é realizada nos construtores da classe base e nos construtores de cada classe herdeira (linhas 27, 46 e 55).

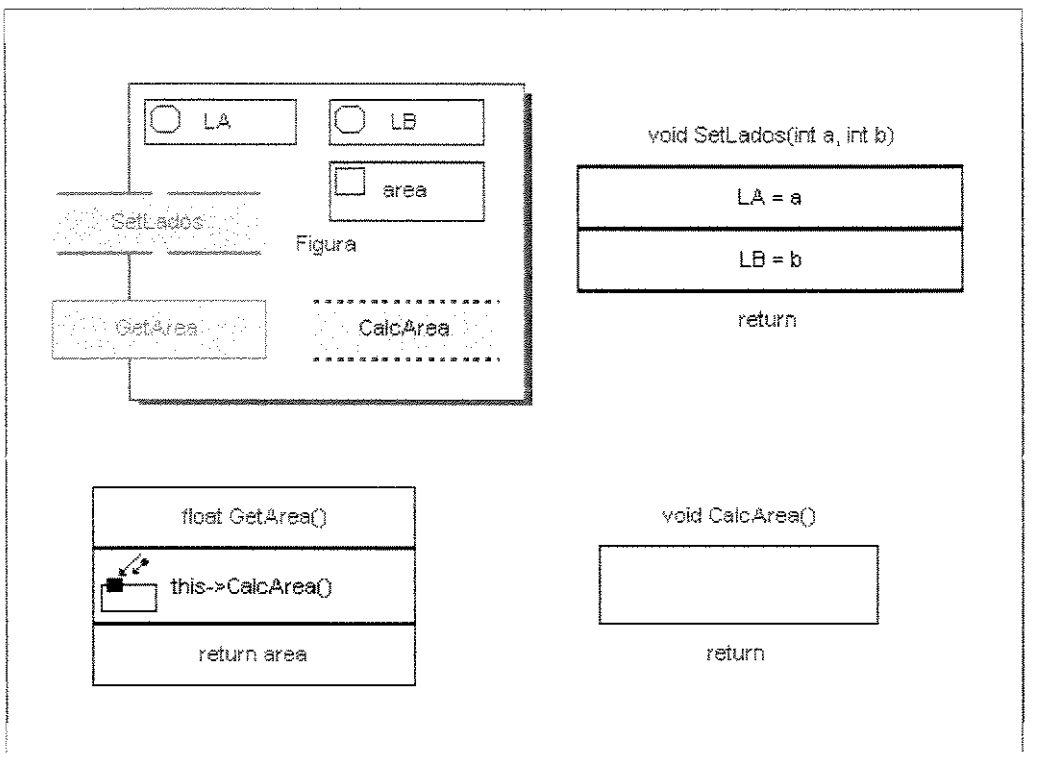

Figura 5.22: A classe *Figura*

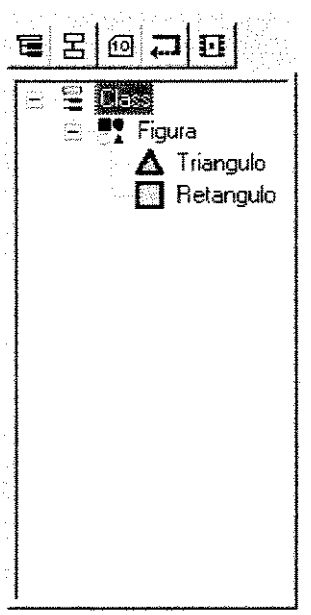

Figura 5.23: A classe *Figura* deriva as classes *Triangulo* e *Retangulo*

Assim, a chamada ao método virtual (linha 37) fica atrelada ao conteúdo do apontador do método, ou seja, o apontador indica a implementação pertinente (linhas 40, 48 e 57) à classe a qual o objeto que recebe a mensagem pertence.

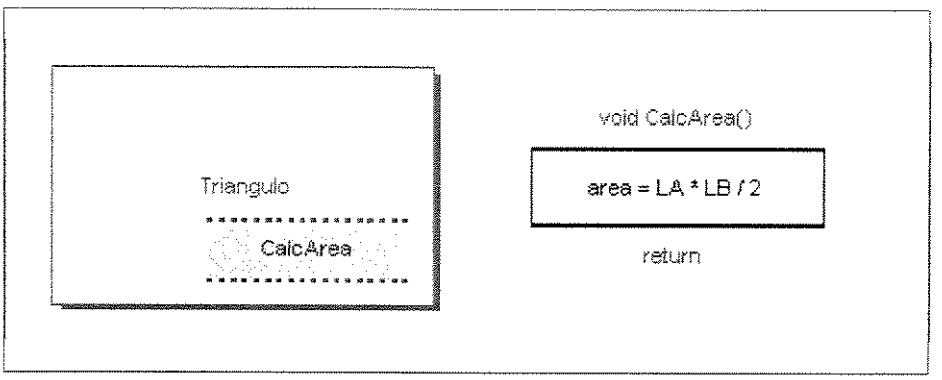

Figura 5.24: A classe *Triangulo*

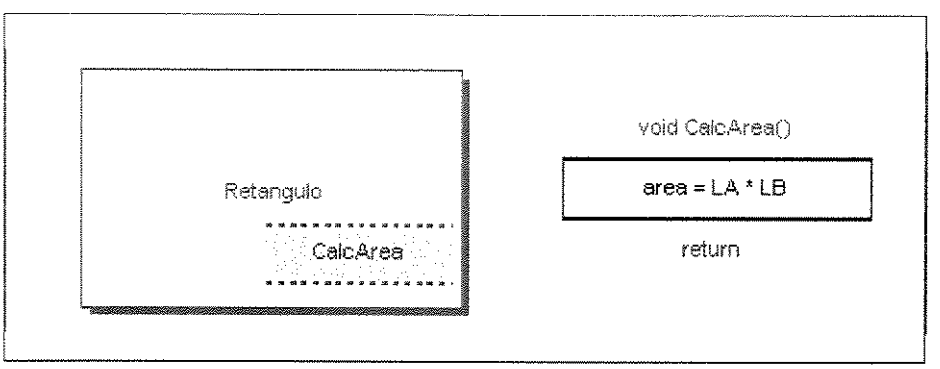

Figura 5.25: A classe Ret*angulo*

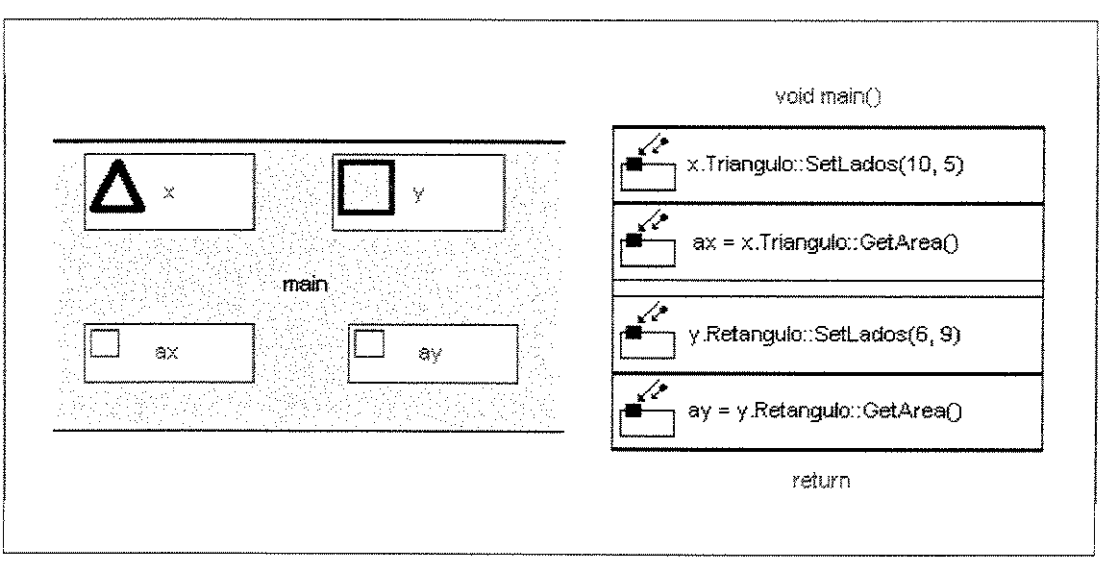

Figura 5.26: O módulo de partida *main*

5.2.9. A biblioteca de funções

A linguagem O++ possui operações que têm equivalência na linguagem C, sendo a tradução realizada na relação de um para um. Porém a maioria das operações da linguagem O++ são traduzidas como uma seqüência de operações em linguagem C, em particular, as operações de mais alto nível e as que manipulam recursos do microcontrolador.

Para facilitar o algoritmo de tradução, foi criada uma biblioteca, em linguagem C, de tipos de dado e códigos utilizados no processo de compilação. Esta biblioteca, que se encontra no anexo V, é constituída por um conjunto de funções usadas para implementar as operações em O++.

Como na programação para microcontroladores há grande preocupação com os requisitos de memória, o compilador O++ apenas anexa ao programa objeto, gerado em C, as funções da biblioteca que são necessárias para a compilação das operações do programa fonte, em linguagem O++. Com isso, diferentemente de muitas linguagens que agregam, ao programa objeto, todas as funções da biblioteca utilizada, o compilador O++ procura gerar o menor programa objeto possível, inserindo nele apenas as funções que implementam as operações do programa fonte, e as referenciadas pelas funções inseridas.

CONCLUSÃO GERAL

Novas metodologias de desenvolvimento de *software*, tais como a programação visual e a programação orientada a objetos, podem ajudar a suprir a demanda por novos produtos baseados em microcontrolador, que ultrapassa as expectativas do mercado. Grande parte dos pré-requisitos e funcionalidades desses produtos são conseguidas pela elaboração de programas, assim o *software* tem sido fundamental para suprir essa demanda.

Com o uso dessas metodologias, a magnitude do código gerado nas aplicações é maior que o das metodologias convencionais. Porém, o ganho de produtividade durante o desenvolvimento e a manutenção é, sem dúvida, um grande atrativo, pois a capacidade de memória e de processamento dos microcontroladores crescem continuamente.

O uso de técnicas visuais pode efetivamente facilitar o desenvolvimento de *software*. Atualmente pode-se observar que um dos maiores sucessos da indústria de *software* tem sido desenvolver ferramentas visuais de apoio à construção de sistemas baseados em linguagens textuais.

A programação orientada a objetos por si só permite um aumento de produtividade aos desenvolvedores de aplicação e quando empregada adequadamente proporciona bons resultados. Porém, ela possui alguns limites, sendo considerada uma evolução, e não uma revolução, no âmbito das metodologias de programação.

O advento da programação visual nos ambientes orientados a objetos permite que se sobreponham alguns limites, através do aumento do número de pessoas que podem projetar e manter suas próprias aplicações. Essas pessoas não são, necessariamente, especializadas em programação, mesmo assim podem montar suas aplicações, através da junção de partes (objetos) devidamente projetadas por especialistas.

O trabalho realizado e, em especial o protótipo desenvolvido, converge para essa linha. Dentro de suas restrições, ele pode contribuir para o desenvolvimento mais produtivo de aplicações microcontroladas e tornar esta tarefa mais acessível a profissionais da área de projeto de sistemas dedicados que não necessariamente detenham um grande conhecimento de metodologias de programação.

Na proposta da linguagem procurou-se implementar as estruturas essenciais para a programação de microcontroladores, usando símbolos intuitivos e auto-explicativos. É

claro que para produzir uma melhor organização desses símbolos, foi feita uma estilização dos mesmos, aproveitando os recursos de cores do monitor de vídeo e introduzindo detalhes que evidenciam as operações representadas por eles.

A escolha da metodologia (POO) e do ambiente de desenvolvimento (Visual C++) se mostrou pertinente. Por um lado, a metodologia ajudou de forma bastante significativa o modelamento do sistema e a própria implementação. A linguagem proposta foi estruturada por elementos independentes (objetos) e por forte comunicação entre eles (mensagens). Por outro lado, os recursos oferecidos pelo ambiente de desenvolvimento possibilitaram a criação de uma interface homem-máquina, cuja simbologia gráfica constitui o alicerce da comunicação entre o computador e o programador. Além disso, o uso do recurso DRAG-DROP permitiu criar um ambiente altamente didático e amigável, uma vez que o usuário "sente o que ele está fazendo".

O algoritmo de tradução difere dos usados nas linguagens tradicionais, que super valorizam as análises sintática e semântica dos programas, já que os mesmos são expressos por textos. Tais análises são realizadas normalmente de forma centralizada, entretanto o algoritmo usado na linguagem proposta por este trabalho procura ressaltar a comunicação entre as operações e os identificadores, gerando o código de forma não centralizada, o que permitiu uma grande diminuição da complexidade do algoritmo de tradução.

Melhorias em versões futuras

O desenvolvimento do protótipo, chamado O++, objetivou comprovar na prática, a possibilidade de agregar os paradigmas da programação orientada a objetos e da programação visual no desenvolvimento de aplicações microcontroladas. Entretanto, o protótipo O++ não é um ambiente robusto e não possui todas as funcionalidades de uma linguagem orientada a objetos disponível comercialmente. Procurou-se utilizar uma grande quantidade de elementos visuais e foram implementados apenas os elementos mais importantes do paradigma da orientação a objetos. Assim, há muito o que se fazer para tornar o O++ um ambiente completo de desenvolvimento. Neste sentido, deve-se ressaltar alguns pontos relativos à herança múltipla, à sobreposição total de nomes; e à auto-referência de classes.

Na versão atual do compilador O++ só foi implementado a herança simples, pois este tipo de herança é a mais usual, na maioria das aplicações. Cada classe herda os atributos e operações de uma única classe pai, entretanto podemos encontrar situações em que uma classe é constituída pela composição de duas ou mais classes distintas. Este tipo de herança, chamada herança múltipla, não é comportado pelo compilador O++.

Pode-se, alternativamente, conseguir utilizar as características de duas ou mais classes através da criação de instâncias dessas classes dentro de uma nova classe. Evidentemente isto não se constitui uma herança múltipla, porém é suficiente para implementar a maioria das situações em que é necessário herdar características de mais de uma classe.

Além da herança múltipla, a linguagem O++ também não suporta a sobreposição total de nomes, ou seja, na versão atual do compilador O++, uma classe pode ter um método com o mesmo nome de um método de outra classe. Todavia, dentro da mesma classe não é permitida esta sobreposição.

A linguagem O++ não suporta também a auto-referência de classes, isto é, uma classe não pode ter como membro de dado uma instância dela mesma. Para contornar esse tipo de situação faz-se necessário a criação de uma classe auxiliar, onde exista um membro de dado da classe que se quer referenciar.

Desdobramentos do trabalho

Um ambiente visual de programação se torna altamente produtivo quando incorpora um simulador em seu quadro de ferramentas. Assim, um componente importante que pode interagir com o compilador O++ é um simulador visual, que através de artifícios gráficos, pode dar uma dimensão mais física ao comportamento dos programas desenvolvidos na linguagem O++. O relacionamento entre objetos pode ser melhor observado se forem introduzidas imagens animadas para ressaltar a dinâmica de comunicação entre eles. Os módulos (ou até mesmo cada operação) podem ser associados a animações que explicitam o algoritmo que está sendo executado. Através de animações, pode haver também uma importante contribuição para que os programadores possam sair da visão estática e burocrática com que os elementos de dados são projetados, permitindo enaltecer os dados manipulados e não vê-los apenas como um mero complemento dos algoritmos.

Um outro possível desdobramento deste trabalho é seu uso como ferramenta de apoio no ensino de programação. O principal propósito do desenvolvimento do protótipo O++ foi verificar a possibilidade de agregar os paradigmas da programação orientada a objetos com a programação visual no aumento de produtividade dos desenvolvedores de aplicações microcontroladas. Entretanto, as características psicológicas levadas em consideração em seu projeto tornaram-no bastante didático, o que permite que ele possa ser utilizado como um bom ponto de partida para a construção de uma ferramenta de apoio ao ensino de programação.

REFERÊNCIAS BIBLIOGRÁFICAS

[ADD97] Addis, T. R., "Modelling and the Human Computer Interface", Proceedings of WorkShop Thinking with Diagrams, Portsmouth/England, January 1997.

[BOO86] Booch, G., "Object-Oriented Development", IEEE Trans. Software Engineering", vol. SE-12-2, February 1989, pp. 221.

[BOT95] Bottoni, Costabile, Levialdi & Mussio, "Formalising Visual Languages", IEEE VL'95", 1995, pp. 45-52.

[BUR94] Burnett, Margaret M., Baker, Marla J., "Classification System for Visual Programming Languages", Techinical Report, Oregon State University, OR/USA, June/1994.

[BUR95] Burnett, Margaret M., et al, *Visual Object-Oriented Programming: Concepts and Environments*, Prentice Hall, 1995.

[CAL94] Calloni, Ben A., Bargert, Donald J., "ICONIC Programming in BACII vs Textual Programming: which is a better Learning Environment ?", SIGCSE Bulletin v. 26 n. 1, Mar. 1994, pp. 188-192.

[CHA90] Chang, Shi-Kuo., et al, *Principles of Visual Programming Systems*, Prentice Hall, 1990.

[CHO97] Chorafas, Dimitris N., *Visual Programming Technology*, McGraw-Hill, 1997.

[COX91] Cox, Brad J., *Programação Orientada a Objetos*, Makron Books, São Paulo/SP, 1991

[DAV83] Davis, Willian S., *Systems Analysis and Design*, Addison-Wesley, 1983

[DIA80] Diaz-Herrera, J. L., Flude, R. C.,"Pascal/HSD: A Graphical Programming System", Proceedings of IEEE Compsac, 1980, pp. 12-26.

[ERW95] Erwig, M., Meyer, B., "Heterogeneous Visual Languages - Integrating Visual and Textual Programming", 11th International IEEE Symposium on Visual Languages, 1995, Darmstadt, Germany.

[GAN92] Ganssle, Jack G., *The Art of Programming Embedded Systems*, Academic Press, 1992

[GRE92] Green, T. R. G., Petre, M., "When Visual Programs are Harder to Read than Textual Programs", Proceedings of ECCE-92: European Conference on Cognitive Ergonomics, Rome/Italy, 1992.

[KRU96] David J. Kruglinski, *Inside Visual C++*, Microsoft Press, 1996.

[LEE95] Leebaert, Derek, et al, *The Future of Software*, MIT Press, 1995.

[NAS84] Nassi, I., Shnneiderman, B., "Flowchart Techniques for Structured Programming", ACM SIGPLAN Notices, vol. 8 n. 8, August 1984, pp. 723-728.

[NAT97] National Semiconductor, COP8Sax Designer's Guide, National Semiconductor, 1997

[NET87] Netto, Samuel Pfromn, *Psicologia da Aprendizagem e do Ensino*, EPU/USP, 1987

[NIC94] Nickerson, Jeffrey Vernon, *Visual Programming*, PhD Dissertation, New York University, September 1994.

[OLI97] Olivier, Patrick, "Diagrams and Machine Reasoning" , Proceedings of WorkShop Thinking with Diagrams, Portsmouth/England, January 1997.

[PAI71] Paivio, Allan, *Imagery and Verbal Process*, HRW, New-York, 1971

[PIC96] Pickering, Chris, Survey of Advanced Technology (1991-1996), Systems Development Inc., Kansas/USA, 1996

[PRE90] Pressman, R. S., *Software Engineering: A Practitioner's Approach*, 3th ed, McGraw-Hill, 1990.

[RIC91] Richard S. Wiener, Lewis J. Pinson, *Programação Orientada para Objeto e C++*, Addison-Wesley/Makron Books, 1991

[SAM97] Samek, Miro, "Portable Inheritance and Polymorphism in C", Embedded Systems Programming Magazine, December 1997.

[SCA88] Scalan, David A., "Should Short Relatively Complex Algorithms be Taught Using Both Graphical and Verbal Methods ?", SIGCSE v. 20 n. 1, Feb. 1988, pp. 185-189

[SCA89] Scalan, David A., "Structured Flowcharts Outperform Pseudocodes: An Experimental Comparison", IEEE Software v. 6 n. 5, Sep. 1989, pp. 28-36

[SCH93] Schultz, Thomas W., *C and the 8051: Programmming and Multtasking*, Prentice Hall, 1993

[SHU88] Shu, Nan C., *Visual Programming*, VNR Company, New-York, 1988

[SHN80] Shneiderman, B., *Software Psychology*, Winthrop Publishers, 1980.

[SIC97] Sickle, Ted Van., *Reusable Software Components - Object-Oriented Embedded Systems Programming in C*, Prentice Hall, 1997.

[SIL98] Silva, M. A. Vieira, Ferreira, E. C., Sousa, A. H. "Modi: A Visual Environment Develop Systems Based on Microcontrollers", Proceedings of *ICMP98: International Conference On Microeletronics and Packaging, Curitiba/Brazil, August 1998, pp. 472-474.*

[SOU95] Sousa, A. H., Ferreira, E. C., *ONAGRO - Um Ambiente Gráfico para Desenvolvimento de Software para Microcontroladore*, Dissertação de Mestrado, FEEC/UNICAMP, Dezembro de 1995.

[SOU96a] Sousa, A. H., Ferreira, E. C., Silva, M. A. Vieira, "ONAGRO - A Graphical Environment to the Development of Microcontrollers Software", Proceedings of WAC-96: Second World Automation Congress, Montpellier/France, May 1996, pp. 301-306.

[SOU96b] Sousa, A. H., Ferreira, E. C., "Implementação de uma Linguagem Icônica para Desenvolvimento de Software para Microcontroladores", SBLP-96: Simpósio Brasileiro de Linguagens de Programação, B. Horizonte, Setembro 1996, pp. 129-139.

[SOU98] Sousa, A. H., Ferreira, E. C., "O++: A Visual Object-Oriented Language for Embedded Systems", Proceedings of *ISSCI-98: International Symposium on Soft Computing for Industry, Achorage/AK/USA, May 1998.*

[SPR85] Springer, S. P., Devtsch, G., *Left Brain, Right Brain*, W. H. Freeman and Company, New York, 1985

[TAP93] Tapscott, Don, Art, Caston, *Paradgm Shift: the New Promise of Information Techonology*, McGraw-Hill, 1993.

[TKA96] Tkach, Daniel, et al, *Visual Modeling Techinique: Object Technology Using Visual Programming*, Addison Wesley, 1996.

[WAS90] Wasserman, A. I., et al, "The Object-Oriented Structured Design Notation for Software Design Representation", IEEE Computer, vol. 23, n. 3, March 1990, pp. 5-0-63.

[WIN91] Winblad, A. L., Edwards, S. D., King, D. R., *Software Orientado ao Objeto.*, Addison-Wesley/Makron Books, 1991

[YOU92] Yourdon, Edward, *Decline & Fall of the American Programmer*, Prentice Hall, 1992

[YOU97] Yourdon, Edward, *Rise & Resurrection of the American Programmer*, Prentice Hall, 1997

ANEXO I

Famílias de Microcontroladores Disponíveis no Mercado

Fabricante	Família	Qtd Bits	Clock da CPU	Máximo de xROM interna	Máximo de RAM externa	Máximo de RAM interna	Registros de propósito geral	Bits de E/S	No. de interru -pções	Qtd de timers e counters	Qtd de portas seriais	Preço US$
Atmel Corporation	AT89C51	8	33MHz	20K	68K	256 bytes	32x8 bits	15-32	8	1-3	1	1-10
	AT89Sxx	8	24 MHz	14K	64K	256 bytes	32x8 bits	32	9	3	2	5-15
	AVR	8	até 16MHz	8K	8M	0-512 bytes	32x8 bits	2-4x8	2	2	1	1.80 1K
Cypress Semiconductor	C47C63XXX	8	6MHz	4K	128-256 bytes	128-256 bytes	-	10-40	10-40	1x14 bits	1	1-4
	C47C64XXX	8	6MHz	8K	256 bytes	256 bytes	-	19-39	19-39	1x14 bits	1	2-8
	C47C66XXX	8	6MHz	8K	256 K	256 bytes	-	19-39	19-39	1x14 bits	2	2-8
Hitachi America Ltd.	H8/300 300L	8	até 16MHz	64K	64K	até 4K	16x8 ou 8x16 bits	82-100	30	3	1-2	3-20 @10k
Integrated Silicon Solutions Inc.	IS80C51/31	8	24MHz	4K	128 bytes	128 bytes	21x8 bits	4x8	5	2x16 bits	1	1.20 @ 1K
	IS80C516	8	6MHz	4K	128 bytes	256 bytes	-	8 + 2x5	3	2x16 bits	0	1 @ 1K
	IS80C52/32	8	40MHz	8K	128 bytes	256 bytes	26x8 bits	4x8	6	3x16 bits	1	1.50 @ 1K
	IS89C52	8	40MHz	8K	128 bytes	256 bytes	26x8 bits	4x8	6	3x8 bits	1	5 @ 1K
Intel Corporation	MCS51/MCS251	8	16MHz	32K	até 64K	até 1K	40x8 bits	4x8	7	2	2	1.50-13 @10k
Microchip Technology Inc.	PIC12Cxxx	8	4-10MHz	3584 bytes	25-128 bytes	25-128 bytes	6x8 bits	6	4	1	0	1.15-2.50 @ 1K
	PIC14Cxxx	8	20MHz	7168 bytes	192 bytes	192 bytes	192x8 bits	20	11	2	0	6.00-7.00 @ 1K

Fabricante	Família	Qtd Bits	Clock da CPU	Máximo de xROM interna	Máximo de RAM externa	Máximo de RAM interna	Registros de propósito geral	Bits de E/S	No. de interru -pções	Qtd de timers e counters	Qtd de portas seriais	Preço US$
	PIC16C5x	8	20MHz	3072 bytes	73 bytes	24-73 bytes	73x8 bits	12-20	-	1	0	1.75-3.25 @ 1K
	PIC16C6x	8	20MHz	4K	384 bytes	128-368 bytes	192x8 bits	13-22	12	1-3	3	3-5 @ 1K
	PIC16C7x	8	20MHz	7168 bytes	192 bytes	128-368 bytes	192x8 bits	12-33	12	2	2	3.50-5.00 @ 1K
	PIC16C92x	8	8MHz	8K	176 bytes	176 bytes	176x8 bits	52	11	3	1	6-8 @ 1K
	PIC16F8x Flash	8	10MHz	2K	68 bytes	68 bytes	68x8 bits	13	4	1	0	2.50-3.50 @ 1K
	PIC17Cxxx	8	33MHz	8K	908 bytes	908 bytes	908 bits	33-50	12	3-4	2	6-8 @ 1K
Mitsubishi	HighSpeed USB MCU	8	24MHz	64K	64K	1K	3x8 bits	61	25	3x8bits, 2x16bits	3	7-10
	M37515	8	8MHz	16K	-	512 bytes	3x8 bits	40	16	4/8 bits	0	-
	M37530	8	8MHz	32K	64K	384K	3x8 bits	29	12	3x8 bits	3	1-2
	M38867	8	8MHz	64K	64K	1K	8x6 bits	72	21	4x8 bits	2	-
	SLIM740	8	8MHz	32K	64K	384 bytes	8x3 bits	29	12	3x8 bits	3	1-1.50
Motorola CSIC/AMCU Division	68HC05 B/X	8	até 8MHz	64K	512 bytes	176-512 bytes	8 bits	32-34	8	2-4	2	4.41-11.81 @50k
	68HC05 C/D	8	até 8MHz	64K	até 512 bytes	176-512 bytes	8 bits	31	9	1-2	2	2.63-5.35 @50k
	68HC05 J/K/P	8	até 8MHz	20K	até 64K	32-192 bytes	8 bits	10-21	5	1-2	1	0.85-3.66 @50k

Fabricante	Família	Qtd Bits	Clock da CPU	Máximo de xROM interna	Máximo de RAM externa	Máximo de RAM interna	Registros de propósito geral	Bits de E/S	No. de interru -pções	Qtd de timers e counters	Qtd de portas seriais	Preço US$
	68HC08	8	até 32MHz	36K	1K	1K	35658 bits	54	10	4	2	4.69-13.13 @50k
Motorola Microcontroller Tech. Group	68HC11 A/E/E20/EA9/811E 2/L	8	2MHz - 3MHz	34K	64K	256-768 bytes	2x8 ou 1x16, 2x16 bits	22-46	22	1x16 bits	2	5-10 @10k
	68HC11 C0/F1/K/KW1/FC0	8	2MHz-4MHz	48K	64K	256-768 bytes	2x8 ou 1x16, 2x16 bits	30-80	22	1x16 bits	2	4.50-13 @10k
	68HC11 D/EC0	8	1MHz-3MHz	8K	64K	192-512 bytes	2x8 ou 1x16 ou 2x16 bits	16-32	22	1x16 bits	2	2.50-7 @10k
	68HC11 KA/KS/P2/PH8/56	8	2MHz-4MHz	48K	64K	1K-2K	2x16 ou 1x16, 2x16 bits	51-62	32	1x16 bits	2-5	8.50-23 @10k
National Semiconductor	COP8	8	até 10MHz	24K	64K	1088 bytes	8x8 bits	40	até 14	3x16 bits	2	-
NEC Electronics	KO Series	8	5MHz	6K	até 2K	216 bytes-2K	32x8 bits	5x8	23	a partir de 5	até 3	2-7
	KOS Series	8	5MHz	16K	6K-14K	128 bytes	8x8 bits	4x8	10	a partir de 3	1-4	1.50-3.00
Philips Semiconductors	80C51	8	10MHz	32K	64K	64 bytes -1K	32x8 bits	19-56	7	2-5	3	1-12 @25k
SGS-Thomson Microelectronics	ST9 PLUS (ST9+)	8	25MHz	62K	2K	2K	224 bits	72	9	5	3	5-15 @ 100K
	ST7231	8	8MHz	48K	768 bytes	768 bytes	8 bits	48	10	2x16 bits timers	2	2.95 @50k
	ST62 65B	8	8MHz	8K	128 bytes	128 bytes	8 bits	21	5	2 timers	1	1.95-2.89 @50k
Sharp Microelectronics Technology	SM8500	8	3MHz - 6MHz	61K	-	1K-2K	-	84	16	1x16 bits, 5x8 bits	2	3.50-5 @10k
Texas Instruments Inc.	TMS370	8	0.5MHz-5MHz	48K	112K	128 bytes-3.5K	128x8-256x8 bits	22-55	13-29	1-3	2	1.25-8 @10k

Fabricante	Família	Qtd Bits	Clock da CPU	Máximo de xROM interna	Máximo de RAM externa	Máximo de RAM interna	Registros de propósito geral	Bits de E/S	No. de interru -pções	Qtd de timers e counters	Qtd de portas seriais	Preço US$
Zilog Inc.	Z8	8	8-16 MHz	4K	64K	60-236 bytes	8 bits	14-32	6	1-2 x12bits	1	-
The Western Design Center Inc.	W65Cxxx	8	8MHz	4K	64K	192bytes	3x8 bits	7x8	4	4x16 bits	1	6-6.40 @10k
Hitachi America Ltd.	H8/300H H85/2000	16	até 20MHz	128K	16M	até 8K	8x32 bits, 16x16 bits, 16x8 bits	79	43	5x16 bits	4-6	6-20 @10k
Intel Corporation	MCS 96	16	16MHz-50MHz	64K	232K	232-1000 bytes	232x16 bits	33-64	15	2	2-4	5-15 @10k
Mitsubishi	H16C	16	16MHz 17MHz	128K	1M	10K	12x16 bits	88	73	8	4	6-9 @50k
	M37702	16	8MHz-25MHz	60K	2K	512-2048 bytes	8x16 bits	68	19	8	até 4	5-8
	M37735	16	12MHz-25MHz	124K	3.9K	3968 bytes	8x16 bits	68	19	8	até 6	5-7
Motorola Microcontroller Tech. Group	68HC12 A4/B32	16	2MHz-8MHz	32K	64K	1K	2x8 ou 1x16, 2x16 bits	63-91	25	1x16 bits	1-3	5.50-18 @10k
	M68HC16	16	16MHz, 20MHz, 25MHz	96K	1M program, 1M data	1K-4K	6x16, 5x20, 1x36, 1x16 bits	46-95	7	3	3	-
	MC68302	16	16MHz - 33MHz	-	16M	1.1K	16, 32 bits	18-37	5	4	6	9-18 @10K
National Semiconductor	HPC16 HPC46	16	20MHz-30MHz	16K	64K	1088 bytes	6x16 bits	52	8	4x16 bits	2	-
NEC Electronics	K4 Series	16	32MHz	256K	até 12K	até 12K	32x16 bits	Até 5	27	4x16	3	4-12
Philips Semiconductors	XA	16	30 MHz	64K	16M	512 bytes	16x21 bits	52	32	3	4	5 @ 1K
SGS-Thomson Microelectronics	ST10	16	20MHz - 25MHz	128K	256K-6M	1K-4K	Nx16 bits	76-111	28-56	5	1-3	8-30 @ 1K

Fabricante	Família	Qtd Bits	Clock da CPU	Máximo de xROM interna	Máximo de RAM externa	Máximo de RAM interna	Registros de propósito geral	Bits de E/S	No. de interru -pções	Qtd de timers e counters	Qtd de portas seriais	Preço US$
Sharp Microelectronics Technology	SM6004	16	20MHz 30MHz	64K	-	2K	6x8 bits	8x1, 16x5	26	6x16	3	10 @10k
The Western Design Center Inc.	W65Cxxx	16	8MHz	8K	16M	576 bytes	35505 bits	8x8	6	8x16	4	12.76-13 @10k
Hitachi America Ltd.	SuperH RISC	32	20MHz - 60MHz	256K	4G	até 8K	8x32 bits	até 106	-	até 3	até 2	8-30 @10k
Motorola Transportation Systems Group	M68300 68F333	32	16MHz, 20MHz, 25MHz	64K	16M	1K-19.5K	18x32, 1x16 bits	46-95	7	-	3	-
Sharp Microelectronics Technology	LH77790	32	16 MHz 25MHz	-	-	2K	32x32 bits	24	12	3x16-bit	3	25 @10k

ANEXO II

Compiladores C Disponíveis no Mercado para Microcontroladores

Empresa	CPU alvo	Plataforma	Assembler incluído	Código ANSI	Recursivo & Reentrante	Manual online	Manual impresso	Otimização por velocidade	Otimização por tamanho	Preço US$
Advanced Micro Computer System	C: 68HC11, 80251, 8051xA,	DOS, Win 3.1	✓	✓			✓	✓	✓	149
Archimedes Software Inc.	C: 8051/XA, 80251, 80166, 68HC05/8/11/12/16, 68K, 683xx	Win 3.1/95/NT	✓	✓	✓	✓	✓	✓	✓	1.995
Avocet Systems	C: 68HC05/08, 68HC11, 68HC16, 280, Z8, 6502, 8051, 8051xA, Z80, 68k/CPU-32, G65816	DOS, Win 3.1/95/NT	✓	✓	✓		✓	✓	✓	de 800 até 995
Byte Craft Limited	C: MELPS740, 68HO8, 68HC05, COP8, PIC 16/17Cxx, Z8	DOS, SunOS, Solaris, HP-UX	✓		✓		✓			de 795 até 1.495
Ceibo Inc.	C: 8051xA	Win 3.1/95/NT	✓	✓		✓	✓		✓	2.395
Cmx Co	C: 68HC05/08, 68HC11, 80251, 80186, 8051xA, 280, 2180, H-8, 8051, 80C16x, 68k/CPU-32	DOS, Win 3.1/95/NT	✓	✓	✓	✓	✓	✓	✓	-
Cosmic Software Inc.	C: 6809, 68HC05/8/11/12/16, 68K, 683xx	DOS, Win 3.1/95/NT, SunOS, Solaris, HP,Unix	✓	✓	✓	✓	✓	✓	✓	de 990 até 1.750
Crossware Products	C: 8051, 68k/CPU-32	Win 95/NT	✓	✓	✓	✓	✓	✓	✓	1.300
Custom Computer Services Inc.	C: PIC,16Cxx	DOS, Win 3.1/95/NT				✓	✓			99
Embedded Performance Inc.	C: MIPS, R3K, R4K, R5K, 29K	DOS, Win 3.1/95/NT, SunOS, Solaris, HP, HP9000, AIX	✓	✓	✓		✓	✓	✓	1.750
Forth Inc.	C: 68HC11, 68HC16, 80186, 8051xA, 68k/CPU-32 DSP: 320Cxx	DOS, Win 95/NT	✓	✓	✓		✓			1.995

Empresa	CPU alvo	Plataforma	Assembler incluído	Código ANSI	Recursivo & Reentrante	Manual online	Manual impresso	Otimização por velocidade	Otimização por tamanho	Preço US$
Franklin Software Inc.	C: 80251, 8051xA, 8051	DOS, Win 3.1/95/NT	✓	✓	✓	✓		✓	✓	1.395
Hitachi America Ltd	C: H-8, Super H	Win 95/NT	✓	✓	✓	✓	✓	✓	✓	de 600 até 900
HIWARE	C: 68HC05/08, 68HC11, 68HC16, 8051xA, 68HC12, ST7, 68k/CPU-32	Win 3.1/95/NT, Solaris	✓	✓	✓	✓	✓	✓	✓	de 760 até 1380
IAR Systems	C: 68HC11, 68HC16, 80251, H-8, 8051, 80196, 80296, Z80, 64180, 68HC12, Melps7700-740, M16C, 78K0, 78K4	DOS, Win 3.1/95/NT, SunOS, Solaris, HP	✓	✓	✓	✓	✓	✓	✓	1.595
Intel Corporation	C: MCS51,MCS251, MCS96	DOS, SunOS, Solaris, HP-UX, AIX	✓	✓	✓		✓	✓	✓	2.000
Introl Corporation	C: 68HC11, 68HC16, 68HC12, 68k/CPU-32	DOS, Win 3.1/95/NT, SunOS, Solaris, AIX, LINUX, NetBSD, HP/UX	✓	✓	✓	✓		✓	✓	2.000
Keil Software	C: 80251, 166/167, 8051	DOS, Win 3.1/95/NT	✓	✓	✓	✓	✓	✓	✓	de 1.595 até 2.395
Micro Computer Control Corp.	C: 8051, Z8	DOS	✓		✓		✓			150
Microchip Technology Inc.	C: PICmicro	DOS, Win 3.1/95	✓	✓	✓	✓	✓		✓	695
Mosaic Industries Inc.	C: 68HC11	Win 3.1/95/NT	✓	✓	✓		✓	✓	✓	375
Motorola	C: 68HC11, 68HC16, 68HC12, 68k/CPU-32	Win 95/NT, SunOS, Solaris, AIX, HP-UX	✓	✓	✓	✓				-

Empresa	CPU alvo	Plataforma	Assembler incluído	Código ANSI	Recursivo & Reentrante	Manual online	Manual impresso	Otimização por velocidade	Otimização por tamanho	Preço US$
Object Software Inc.	C: 68HC11, 386/486/Pentium, 68k/CPU-32, PowerPC	Win 3.1/95/NT	✓	✓	✓	✓	✓	✓	✓	495
Production Languages Corp.	C: 8051, 80251, 8x93xAx, 8x93xHx, z8 family, DSP: Z89462, Z893xx	Win 3.1/95	✓	✓	✓	✓	✓	✓	✓	de 495 até 1.995
Rigel Corporation	C: 8051xA DSP: SABC166, ST10	Win 3.1/95/NT	✓			✓	✓	✓	✓	de 100 até 295
Sierra Systems	C: p51XA, 68K, 683xx	DOS, Win 95/NT	✓	✓	✓		✓	✓	✓	2.000
Softools Inc.	C: 280, 2180, 8085	DOS, Win 3.1/95/NT	✓	✓	✓		✓	✓	✓	699
Tasking	C: 80251, 8051xA, 80C166, 8051, 196, 296, TLCS-900 SMC88	DOS, Win 3.1/95/NT, SunOS, Solaris, HP-UX	✓	✓	✓	✓	✓	✓	✓	1.995
Texas Instruments Inc.	C: TMS370	DOS, Win 3.1/95/NT, SunOS, Solaris, HP9000	✓	✓	✓	✓	✓	✓	✓	de 1.500 até 3.500
The Western Design Center Inc.	C: W65C816S	DOS		✓	✓	✓	✓			99
Zilog Inc.	C: ZB, DSP: Z89C00, Z893xx, Z87000, Z87010, Z89462	Win 3.1/95	✓	✓	✓	✓	✓	✓	✓	de 700 até 1.850

ANEXO III

Tabela de códigos fontes C/C++ usados pelo compilador O++

```
SrcCodC[C_NULO] = ``;

// MAIN DEFINE
SrcCodC[C_MAIN_DEF]   = `<TYPE> main (`;

// INTERRUPT DEFINE
SrcCodC[C_INTR_DEF]   = `interrupt void <METHOD> (void`;

// CLASS DEFINE
SrcCodC[C_CLASS_DEF]   = `#define _<CLASS>`;

// CLASS DEFINE WITH FATHER
SrcCodC[C_CLASS_DEF_FATHER]   = `#define _<CLASS> _<CLASS>`;

// CLASS TYPEDEF
SrcCodC[C_CLASS_TYPE]   = `typedef struct { _<CLASS> } <CLASS>;`;

// CONSTRUCTOR DEFINE
SrcCodC[C_CONSTR_DEF]   = `void _<CLASS>_ (<CLASS>* this`;

// ATRIB. METODO POLIFORMICO
SrcCodC[C_ATRIB_MET_VIRTUAL]   = `this->_<METHOD> = <CLASS>_<METHOD>;`;

// CHAMADA AO CONSTRUTOR
SrcCodC[C_CONSTR_CALL]   = `_<CLASS>_ ((<CLASS>*)&<OBJ>`;

// MEMBER DEFINE
SrcCodC[C_MEMBER_DEF]   = `<TYPE> <MEMBER>`;

// VIRTUAL MEMBER DEFINE
SrcCodC[C_VIRTUAL_MEMBER_DEF]   = `<TYPE> (*_<MEMBER>)();`;
// OBJECT DEFINE
SrcCodC[C_OBJ_DEF]   = `<CLASS> <OBJ>;`;
```

```
// HEADER METHOD AND FUNCTION DEFINE
SrcCodC[C_HEADER_METHOD_DEF]   = `<TYPE> _<CLASS>_<METHOD> (<CLASS>* this`;
SrcCodC[C_HEADER_FUNCTION_DEF]   = `<TYPE> _<FUNCTION> (<INPUT>`;

// METHOD AND FUNCTION DEFINE
SrcCodC[C_METHOD_DEF]   = `<TYPE> _<CLASS>_<METHOD> (<CLASS>* this`;
SrcCodC[C_FUNCTION_DEF]   = `<TYPE> _<FUNCTION> (<INPUT>`;

// METHOD AND FUNCTION CALL
SrcCodC[C_LOCAL_METHOD_CALL]   = `_<CLASS>_<METHOD> (<OBJ>`;
SrcCodC[C_METHOD_CALL]   = `_<CLASS>_<METHOD> ((<CLASS>*)&<OBJ>`;
SrcCodC[C_FUNCTION_CALL]   = `_<FUNCTION> (`;
SrcCodC[C_VIRTUAL_CALL]   = `this->_<METHOD> (this`;

// MEMBER ACCESS
SrcCodC[C_MEMBER_ACCESS]   = `this-><MEMBER>`;

// SET VALUE
SrcCodC[C_SET_VALUE]   = `<ID> = <VL>`;

// SET INTERRUPT VECTOR
SrcCodC[C_SET_VECTOR]   = `set_vector(<TYPE>, <METHOD>);`;

// TIMER INI
SrcCodC[C_TIMER_INI0]   = `TMOD=<VL>;`;
SrcCodC[C_TIMER_INI1]   = `TR<IND>=<VL>; `;
SrcCodC[C_TIMER_INI2]   = `TL<IND>=<VL>; TH<IND>=<VL>;`;

// SERIAL INI
SrcCodC[C_SERIAL_INI]   = `SCON=<VL>; TMOD=TMOD & 0x2F; TH1=<VL>; TR1=1;`;
// FLAG INTR ENABLE
SrcCodC[C_FLAG_INTR_EX0]   = `EX0`;
SrcCodC[C_FLAG_INTR_EX1]   = `EX1`;
SrcCodC[C_FLAG_INTR_ET0]   = `ET0`;
```

```
SrcCodC[C_FLAG_INTR_ET1]   = `ET0`;
SrcCodC[C_FLAG_INTR_ES]    = `ES`;
SrcCodC[C_FLAG_INTR_EA]    = `EA`;

// FLAG INTR PRIORITY
SrcCodC[C_FLAG_INTR_PX0]   = `PX0`;
SrcCodC[C_FLAG_INTR_PX1]   = `PX1`;
SrcCodC[C_FLAG_INTR_PT0]   = `PT0`;
SrcCodC[C_FLAG_INTR_PT1]   = `PT0`;
SrcCodC[C_FLAG_INTR_PS]    = `PS`;

// FLAG INTR ACTIVATION

SrcCodC[C_FLAG_INTR_IT0]   = `IT0`;
SrcCodC[C_FLAG_INTR_IT1]   = `IT1`;

// PORT BITS
SrcCodC[C_P0_BITS]   = `P0_BITS.B<IND>`;
SrcCodC[C_P1_BITS]   = `P1_BITS.B<IND>`;
SrcCodC[C_P2_BITS]   = `P2_BITS.B<IND>`;
SrcCodC[C_P3_BITS]   = `P3_BITS.B<IND>`;

// READ PORT BITS

SrcCodC[C_READ_PORT_BIT]   = `<ID>._BIT.B<IND> = <VL>_BITS.B<IND>;`;

// READ PORT BITS
SrcCodC[C_WRITE_PORT_BIT]   = `<VL>_BITS.B<IND> = <ID>._BIT.B<IND>;`;

// DEFINE PORT
SrcCodC[C_DEF_PORT]   = `_defPort <ID>;`;
// DEFINE PORT2
SrcCodC[C_DEF_PORT2]   = `#define <ID> <VL>`;
```

```
// DEFINE MEM. MAP. PORT
SrcCodC[C_DEF_MEM_MAP_PORT]   = `_unionBitByte <ID>;`;

// DEFINE MEM. MAP. PORT2
SrcCodC[C_DEF_MEM_MAP_PORT2]   = `_defMemMapPort <ID> @<VL>;`;

// DEFINE FACE PORT
SrcCodC[C_DEF_PORT_FACE]   = `_defMemMapPort _ADDR<VL> @<VL>; _unionBitByte
_FACE<VL>;`;

// READ FACE (_bitByteAux) BITS
SrcCodC[C_READ_FACE_BIT]   = `<ID>._BIT.B<IND> = _FACE<VL>._BIT.B<IND>;`;

// WRITE FACE (_bitByteAux) BITS
SrcCodC[C_WRITE_FACE_BIT]   = `_FACE<VL>._BIT.B<IND> = <ID>._BIT.B<IND>;`;

// GET FACE (_bitByteAux) BYTE
SrcCodC[C_GET_FACE]   = `_ADDR<VL> = _FACE<VL>._BYTE`;

// SET FACE (_bitByteAux) BYTE
SrcCodC[C_SET_FACE]   = `_FACE<VL>._BYTE = _ADDR<VL>`;


//---------------- final de arquivo --------------------
```

ANEXO IV

Exemplo de código gerado em C que utiliza métodos virtuais

```
1   //--------------Head Includes---------------
2   #include "C_LIB.H"
3   //--------------Lib Functions---------------
4   //--------------Class Defines---------------
5   #define _Figura\
6           int LA;\
7           int LB;\
8           float area;\
9   void (*_CalcArea)();
10  typedef struct { _Figura } Figura;
11  void _Figura_SetLados (Figura* this, int a, int b);
12  float _Figura_GetArea (Figura* this);
13  void _Figura_CalcArea (Figura* this);
14  #define _Triangulo _Figura
15  typedef struct { _Triangulo } Triangulo;
16  void _Triangulo_CalcArea (Triangulo* this);
17  #define _Retangulo _Figura
18  typedef struct { _Retangulo } Retangulo;
19  void _Retangulo_CalcArea (Retangulo* this);
20  //------------Functions Prototypes--------------
21  //------------User Define Ports---------------
22  //------------Port Faces---------------
23  //--------------Global Variables----------------
24  //--------------Method Defines----------------
25  void _Figura_ (Figura* this)
26  {
27          this->_CalcArea = _Figura_CalcArea;
28  }
29  void _Figura_SetLados (Figura* this, int a, int b)
30  {
31          this->LA =a;
```

```
32          this->LB =b;
33          return ;
34  }
35  float _Figura_GetArea (Figura* this)
36  {
37          this->_CalcArea (this);
38          return this->area;
39  }
40  void _Figura_CalcArea (Figura* this)
41  {
42          return ;
43  }
44  void _Triangulo_ (Triangulo* this)
45  {
46          this->_CalcArea = _Triangulo_CalcArea;
47  }
48  void _Triangulo_CalcArea (Triangulo* this)
49  {
50          this->area =this->LA *this->LB /2;
51          return ;
52  }
53  void _ Retangulo _ (Triangulo* this)
54  {
55          this->_CalcArea = _ Retangulo _CalcArea;
56  }
57  void _Retangulo_CalcArea (Retangulo* this)
58  {
59          this->area =this->LA *this->LB;
60          return ;
61  }
62  //----------VECTORS----------
```

```
63  //----------Functions Bodies----------
64  //----------MAIN----------
65  void main ()
66  {
67          Triangulo x;
68          float ax;
69          Retangulo y;
70          float ay;
71          _Triangulo_ ((Triangulo*)&x);
72  /*---------- STARTUP CODE ----------*/
73  /*---------- END STARTUP CODE ----------*/
74          _Figura_SetLados ((Figura*)&x, 10, 5);
75          ax = _Figura_GetArea ((Figura*)&x);
76          _Figura_SetLados ((Figura*)&y, 6, 9);
77          ay = _Figura_GetArea ((Figura*)&y);
78          return ;
79  }
```

ANEXO V

Biblioteca usada para implementar as operações em O++

```
/* ----- Cabecalhos ------*/
#include <8051.h>
#include <intrpt.h>

/* ---- Parametros da CPU ----*/
#define _CPU_CLOCK 6.144  /* Em MHz */
#define _TIME_STEP 0.001  /* Em segundos */
/* VL = (_TIME_STEP/((1.0/_CPU_CLOCK)/12.0) */

/* ---- Definicao de tipos ----*/
typedef union
{
     SFR_BITS _BIT;
     unsigned char _BYTE;
} _unionBitByte;

typedef _unionBitByte _defPort;

typedef volatile far unsigned char _MemAddr;

typedef _MemAddr _defMemMapPort;

/*----- Operacoes com vetores tipo INT -----*/
#ifndef _NSCHARTABINI
#define _NSCHARTABINI                              \
void _nsCharTabIni(char* tab, int len, char vl) \
{                                                  \
   int i;                                           \
   for (i=0;i<len;i++)                               \
      tab[i] = vl;                                  \
}
#endif

#ifndef _NSCHARTABPULSE
#define _NSCHARTABPULSE                        \
void _nsCharTabPulse(char* tab, int len)\
{                                       \
   int i; int j = len/3;                 \
   for (i=0;i<j;i++)                    \
      tab[i] = 0;                       \
   for (i=j;i<len-j;i++)                 \
      tab[i] = 1;                       \
   for (i=len-j;i<len;i++)                \
      tab[i] = 0;                       \
}
#endif
```

```
#ifndef _NSCHARTABSINE
#define _NSCHARTABSINE                          \
void _nsCharTabSine(char* tab, int len) \
{                                               \
   float inc = 6.283185307/len;                 \
   float x = 0.0;                               \
   int i;                                       \
   for (i=0; i<len; i++)                        \
   {                                            \
      tab[i] = (int)(32767*sin(x));             \
      x = x + inc;                              \
   }                                            \
}
#endif
#ifndef _NSCHARTABRAND
#define _NSCHARTABRAND                          \
void _nsCharTabRand(char* tab, int len) \
{                                               \
   int i;                                       \
   for (i=0;i<len;i++)                          \
      tab[i] = rand();                          \
}
#endif

#ifndef _NSCHARTABSUM
#define _NSCHARTABSUM                           \
float _nsCharTabSum(char* tab, int len) \
{                                               \
   int i;                                       \
   float sum = 0.0;                             \
   for (i=0;i<len;i++)                          \
      sum += tab[i];                            \
   return sum;                                  \
}
#endif

#ifndef _NSCHARTABMEAN
#define _NSCHARTABMEAN                          \
float _nsCharTabMean(char* tab, int len)\
{                                               \
   int i;                                       \
   float sum = 0.0;                             \
   for (i=0;i<len;i++)                          \
      sum += tab[i];                            \
   return (sum/len);                            \
}
#endif
```

```
#ifndef _NSCHARTABMAX
#define _NSCHARTABMAX                          \
int _nsCharTabMax(char* tab, int len)     \
{                                          \
   int i;                                  \
   int max = -128;                          \
   for (i=0;i<len;i++)                       \
      if(tab[i] > max)                      \
         max = tab[i];                      \
   return max;                               \
}
#endif

#ifndef _NSCHARTABMIN
#define _NSCHARTABMIN                          \
int _nsCharTabMin(char* tab, int len)     \
{                                          \
   int i;                                  \
   int min = 127;                           \
   for (i=0;i<len;i++)                       \
      if(tab[i] < min)                      \
         min = tab[i];                      \
   return min;                               \
}
#endif


#ifndef _NSCHARTABASCSORT
#define _NSCHARTABASCSORT                        \
void _nsCharTabAscSort(char* tab, int len)     \
{                                               \
   int ip=len-1;                                 \
   int i;                                       \
   char aux;                                    \
   char fim;                                    \
   do                                           \
   {                                            \
      fim = 1;                                   \
      for (i=0;i<ip;i++)                            \
      {                                         \
         if(tab[i] > tab[i+1])                      \
         {                                      \
            aux = tab[i];                          \
            tab[i] = tab[i+1];                      \
            tab[i+1] = aux;                         \
            fim = 0;                             \
```

```
            }                                   \
        }                                       \
        ip--;                                    \
    } while (!fim);                                 \
}
#endif

#ifndef _NSCHARTABDESSORT
#define _NSCHARTABDESSORT                              \
void _nsCharTabDesSort(char* tab, int len)      \
{                                               \
    int ip=len-1;                                  \
    int i;                                        \
    char aux;                                     \
    char fim;                                     \
    do                                             \
    {                                             \
        fim = 1;                                   \
        for (i=0;i<ip;i++)                            \
        {                                         \
            if(tab[i] < tab[i+1])                     \
            {                                     \
                aux = tab[i];                        \
                tab[i] = tab[i+1];                    \
                tab[i+1] = aux;                       \
                fim = 0;                           \
            }                                     \
        }                                         \
        ip--;                                      \
    } while (!fim);                                 \
}
#endif

#ifndef _NSCHARTABSEARCH
#define _NSCHARTABSEARCH                                    \
int _nsCharTabSearch(char vl, char* tab, int len)    \
{                                                       \
    int i=0;                                             \
    int ret = -1;                                           \
    while ((ret == -1) && (i<len))                             \
    {                                                       \
        if (tab[i] == vl)                                      \
            ret = i;                                     \
        i++;                                                \
    }                                                       \
    return ret;                                                \
}
```

```
#endif

/*----- Operacoes com vetores tipo INT -----*/
#ifndef _NSINTTABINI
#define _NSINTTABINI                                \
void _nsIntTabIni(int* tab, int len, int vl)     \
{                                                  \
   int i;                                             \
   for (i=0;i<len;i++)                                 \
      tab[i] = vl;                                    \
}
#endif

#ifndef _NSINTTABPULSE
#define _NSINTTABPULSE                          \
void _nsIntTabPulse(int* tab, int len)     \
{                                          \
   int i; int j = len/3;                        \
   for (i=0;i<j;i++)                          \
      tab[i] = 0;                             \
   for (i=j;i<len-j;i++)                        \
      tab[i] = 1;                             \
   for (i=len-j;i<len;i++)                        \
      tab[i] = 0;                             \
}
#endif

#ifndef _NSINTTABSINE
#define _NSINTTABSINE                           \
void _nsIntTabSine(int* tab, int len)     \
{                                          \
   float inc = 6.283185307/len;                 \
   float x = 0.0;                             \
   int i;                                   \
   for (i=0; i<len; i++)                        \
   {                                       \
      tab[i] = (int)(32767*sin(x));        \
      x = x + inc;                         \
   }                                       \
}
#endif

#ifndef _NSINTTABRAND
#define _NSINTTABRAND                           \
void _nsIntTabRand(int* tab, int len)     \
{                                          \
   int i;                                        \
```

```
   for (i=0;i<len;i++)                    \
      tab[i] = rand();                   \
}
#endif

#ifndef _NSINTTABSUM
#define _NSINTTABSUM                      \
float _nsIntTabSum(int* tab, int len)     \
{                                         \
   int i;                                 \
   float sum = 0.0;                       \
   for (i=0;i<len;i++)                    \
      sum += tab[i];                      \
   return sum;                            \
}
#endif

#ifndef _NSINTTABMEAN
#define _NSINTTABMEAN                     \
float _nsIntTabMean(int* tab, int len)    \
{                                         \
   int i;                                 \
   float sum = 0.0;                       \
   for (i=0;i<len;i++)                    \
      sum += tab[i];                      \
   return (sum/len);                      \
}
#endif

#ifndef _NSINTTABMAX
#define _NSINTTABMAX                      \
int _nsIntTabMax(int* tab, int len)       \
{                                         \
   int i;                                 \
   int max = -32768;                      \
   for (i=0;i<len;i++)                    \
      if(tab[i] > max)                    \
         max = tab[i];                    \
   return max;                            \
}
#endif

#ifndef _NSINTTABMIN
#define _NSINTTABMIN                      \
int _nsIntTabMin(int* tab, int len)       \
{                                         \
   int i;                                 \
```

```
   int min = 32767;                    \
   for (i=0;i<len;i++)                   \
      if(tab[i] < min)                 \
         min = tab[i];                 \
   return min;                          \
}
#endif

#ifndef _NSINTTABASCSORT
#define _NSINTTABASCSORT                          \
void _nsIntTabAscSort(int* tab, int len)     \
{                                          \
   int ip=len-1;                             \
   int i, aux;                               \
   char fim;                                \
   do                                       \
   {                                        \
      fim = 1;                              \
      for (i=0;i<ip;i++)                       \
      {                                     \
         if(tab[i] > tab[i+1])                 \
         {                                  \
            aux = tab[i];                     \
            tab[i] = tab[i+1];                \
            tab[i+1] = aux;                   \
            fim = 0;                        \
         }                                  \
      }                                     \
      ip--;                                 \
   } while (!fim);                             \
}
#endif

#ifndef _NSINTTABDESSORT
#define _NSINTTABDESSORT                          \
void _nsIntTabDesSort(int* tab, int len)     \
{                                          \
   int ip=len-1;                             \
   int i, aux;                               \
   char fim;                                \
   do                                       \
   {                                        \
      fim = 1;                              \
      for (i=0;i<ip;i++)                       \
      {                                     \
         if(tab[i] < tab[i+1])                 \
         {                                  \
```

```
            aux = tab[i];                \
            tab[i] = tab[i+1];             \
            tab[i+1] = aux;                \
            fim = 0;                     \
         }                               \
      }                                  \
      ip--;                               \
   } while (!fim);                          \
}
#endif

#ifndef _NSINTTABSEARCH
#define _NSINTTABSEARCH                                  \
int _nsIntTabSearch(int vl, int* tab, int len)    \
{                                                \
   int i=0;                                       \
   int ret = -1;                                    \
   while ((ret == -1) && (i<len))                      \
   {                                              \
      if (tab[i] == vl)                             \
         ret = i;                                 \
      i++;                                        \
   }                                              \
   return ret;                                       \
}
#endif

/*----- Operacoes com vetores tipo REAL -----*/
#ifndef _NSREALTABINI
#define _NSREALTABINI                                    \
void _nsRealTabIni(float* tab, int len, float vl)     \
{                                                     \
   int i;                                                \
   for (i=0;i<len;i++)                                    \
      tab[i] = vl;                                      \
}
#endif
#ifndef _NSREALTABPULSE
#define _NSREALTABPULSE                                \
void _nsRealTabPulse(float* tab, int len)     \
{                                                \
   int i; int j = len/3;                            \
   for (i=0;i<j;i++)                                 \
      tab[i] = 0.0;                                \
   for (i=j;i<len-j;i++)                              \
      tab[i] = 1.0;                                \
   for (i=len-j;i<len;i++)                             \
```

```
      tab[i] = 0.0;                          \
}
#endif

#ifndef _NSREALTABSINE
#define _NSREALTABSINE                              \
void _nsRealTabSine(float* tab, int len)     \
{                                                \
   float inc = 6.283185307/len;                \
   float x = 0.0;                               \
   int i;                                      \
   for (i=0; i<len; i++)                        \
   {                                           \
      tab[i] = sin(x);                          \
      x = x + inc;                             \
   }                                           \
}
#endif

#ifndef _NSREALTABRAND
#define _NSREALTABRAND                              \
void _nsRealTabRand(float* tab, int len)     \
{                                                \
   int i;                                                \
   for (i=0;i<len;i++)                                    \
      tab[i] = 1.0 * rand();                    \
}
#endif

#ifndef _NSREALTABSUM
#define _NSREALTABSUM                               \
float _nsRealTabSum(float* tab, int len)     \
{                                                \
   int i;                                       \
   float sum = 0.0;                             \
   for (i=0;i<len;i++)                             \
      sum += tab[i];                            \
   return sum;                                  \
}
#endif
#ifndef _NSREALTABMEAN
#define _NSREALTABMEAN                              \
float _nsRealTabMean(float* tab, int len)     \
{                                                \
   int i;                                       \
   float sum = 0.0;                             \
   for (i=0;i<len;i++)                             \
```

```
      sum += tab[i];                          \
   return (sum/len);                          \
}
#endif
#ifndef _NSREALTABMAX
#define _NSREALTABMAX                              \
float _nsRealTabMax(float* tab, int len)     \
{                                             \
   int i;                                      \
   float max = 3.4e-38;                         \
   for (i=0;i<len;i++)                           \
      if(tab[i] > max)                          \
         max = tab[i];                          \
   return max;                                  \
}
#endif

#ifndef _NSREALTABMIN
#define _NSREALTABMIN                              \
float _nsRealTabMin(float* tab, int len)     \
{                                             \
   int i;                                      \
   float min = 3.4e38;                          \
   for (i=0;i<len;i++)                           \
      if(tab[i] < min)                          \
         min = tab[i];                          \
   return min;                                  \
}
#endif

#ifndef _NSREALTABASCSORT
#define _NSREALTABASCSORT                          \
void _nsRealTabAscSort(float* tab, int len)     \
{                                             \
   int ip=len-1;                                \
   int i, aux;                                  \
   char fim;                                    \
   do                                           \
   {                                            \
      fim = 1;                                  \
      for (i=0;i<ip;i++)                        \
      {                                         \
         if(tab[i] > tab[i+1])                  \
         {                                      \
            aux = tab[i];                       \
            tab[i] = tab[i+1];                  \
            tab[i+1] = aux;                     \
```

```
            fim = 0;                          \
         }                                    \
      }                                       \
      ip--;                                    \
   } while (!fim);                                \
}
#endif

#ifndef _NSREALTABDESSORT
#define _NSREALTABDESSORT                              \
void _nsRealTabDesSort(float* tab, int len)     \
{                                                \
   int ip=len-1;                                  \
   int i, aux;                                     \
   char fim;                                      \
   do                                              \
   {                                               \
      fim = 1;                                     \
      for (i=0;i<ip;i++)                              \
      {                                            \
         if(tab[i] < tab[i+1])                        \
         {                                         \
            aux = tab[i];                          \
            tab[i] = tab[i+1];                       \
            tab[i+1] = aux;                          \
            fim = 0;                               \
         }                                         \
      }                                            \
      ip--;                                        \
   } while (!fim);                                 \
}
#endif

#ifndef _NSREALTABSEARCH
#define _NSREALTABSEARCH                                    \
int _nsRealTabSearch(float vl, float* tab, int len)    \
{                                                       \
   int i=0;                                             \
   int ret = -1;                                          \
   while ((ret == -1) && (i<len))                            \
   {                                                    \
      if (tab[i] == vl)                                    \
         ret = i;                                       \
      i++;                                              \
   }                                                    \
   return ret;                                             \
}
```

```
#endif

/*----- Operacoes com a placa ugh -----*/
#ifndef _NSINTBINHEX
#define _NSINTBINHEX                          \
void _nsIntBinHex(unsigned int source, char* target)    \
{                                             \
   static char tab[]="0123456789ABCDEF";      \
   char ind;                                  \
   char digit;                                \
   char ref;                                  \
   if (source < 256)                          \
      ref = 2;                                \
   else                                       \
      ref = 4;                                \
   for (ind=0;ind<ref;ind++)                  \
   {                                          \
      digit = source & 0x000F;                \
      source = source >> 4;                   \
      target[(ref-1) - ind] = tab[digit];     \
   }                                          \
   target[ind] = '\0';                        \
}
#endif

#ifndef _UGHCHARPRINT
#define _UGHCHARPRINT                         \
char _ughCharValue;                           \
void _ughCharPrint(char vl)                   \
{                                             \
   _ughCharValue = vl;                        \
   asm(" mov dptr, #__ughCharValue ");        \
   asm(" movx a,@dptr ");                     \
   asm(" lcall 0041h ");                      \
}
#endif

#ifndef _UGHSTRINGPRINT
#define _UGHSTRINGPRINT                       \
void _ughStringPrint(char* tab)               \
{                                             \
   while (*tab)                               \
   {                                          \
      _ughCharPrint(*tab);                    \
      tab++;                                  \
   }                                          \
   asm(" lcall 004ah ");                      \
```

```
}
#endif

#ifndef _UGHBYTEPRINT
#define _UGHBYTEPRINT                          \
char _byteTarget[3];                             \
void _ughBytePrint(unsigned char vl)             \
{                                               \
   _nsIntBinHex((unsigned int)vl, _byteTarget);          \
   _ughStringPrint(_byteTarget);                 \
}
#endif

#ifndef _UGHINTPRINT
#define _UGHINTPRINT                         \
char _intTarget[5];                            \
void _ughIntPrint(unsigned int vl)                    \
{                                             \
   _nsIntBinHex(vl, _intTarget);                \
   _ughStringPrint(_intTarget);                \
}
#endif


/*----- Operacoes com a porta de comunicacao serial -----*/
#ifndef _NSCHARSERIALPUT
#define _NSCHARSERIALPUT              \
void _nsCharSerialPut(char vl)             \
{                                                \
     while(!TI)                                  \
          continue;                         \
     TI = 0;                                     \
     SBUF = vl;                                  \
}
#endif

#ifndef _NSBUFFERSERIALPUT
#define _NSBUFFERSERIALPUT                                  \
void _nsBufferSerialPut(char* buf, int qtd)          \
{                                                                \
     int i;                                                      \
     for (i=0;i<qtd;i++)                                \
     {                                                           \
          _nsCharSerialPut(*buf);                        \
          buf++;                                                 \
   }                                                             \
}
```

```
#endif

#ifndef _NSSTRINGSERIALPUT
#define _NSSTRINGSERIALPUT                                    \
void _nsStringSerialPut(char* tab)                    \
{                                                                  \
   while (*tab)                                                    \
   {                                                               \
      _nsCharSerialPut(*tab);                          \
      tab++;                                                       \
   }                                                               \
}
#endif

#ifndef _NSCHARSERIALGET
#define _NSCHARSERIALGET                    \
char _nsCharSerialGet (void)                \
{                                                    \
     while(!RI)                                      \
          continue;                                  \
     RI = 0;                                         \
     return SBUF;                                    \
}
#endif

#ifndef _NSBUFFERSERIALGET
#define _NSBUFFERSERIALGET                                          \
void _nsBufferSerialGet(char* buf, int qtd)          \
{                                                                        \
     int i;                                                              \
     for (i=0;i<qtd;i++)                                                 \
     {                                                                   \
          *buf = _nsCharSerialGet();                                     \
          buf++;                                                         \
   }                                                                     \
}
#endif

#ifndef _NSSTRINGSERIALGET
#define _NSSTRINGSERIALGET                                    \
void _nsStringSerialGet(char* tab)                    \
{                                                                  \
   while (*tab)                                                    \
   {                                                               \
      *tab = _nsCharSerialGet();                       \
      tab++;                                                       \
   }                                                               \
```

```
}
#endif

#ifndef _NSTIMEREVENT
#define _NSTIMEREVENT                                          \
unsigned int* _nsTimerEventVal;                                \
void interrupt _nsTimerEventIsr(void)                          \
{                                                                  \
     *_nsTimerEventVal = *_nsTimerEventVal + 1;                \
}                                                                  \
void _nsTimerEventOn(unsigned int* counter)          \
{                                                                  \
     TMOD = TMOD & 0xF5;                                       \
     _nsTimerEventVal = counter;                               \
     *_nsTimerEventVal = 0;                                        \
     set_vector(TIMER0, _nsTimerEventIsr);                     \
     PT0=1; ET0=1; EA=1; TR0=1;                                \
}                                                                  \
void _nsTimerEventOff(void)                                    \
{                                                                  \
     PT0=0; ET0=0; TR0=0;                                      \
}
#endif

#ifndef _NSTIMERPULSE
#define _NSTIMERPULSE                                          \
unsigned char _nsTimerPulseLow;                                \
unsigned char _nsTimerPulseHigh;                     \
unsigned char _nsTimerPulseLi;                                 \
unsigned char _nsTimerPulseHi;                                 \
void interrupt _nsTimerPulseIsr(void)                \
{                                                                  \
     if (_nsTimerPulseLow == 0)                                \
     {                                                             \
          P1_BITS.B0 = 1;                                          \
          if (_nsTimerPulseHigh == 0)                      \
          {                                                        \
               P1_BITS.B0 = 0;                                     \
               _nsTimerPulseHigh = _nsTimerPulseHi;\
               _nsTimerPulseLow = _nsTimerPulseLi; \
          }                                                        \
          else                                                \
               _nsTimerPulseHigh--;                           \
     }                                                             \
     else                                                    \
          _nsTimerPulseLow--;                                \
     TL0 = 0x00; /* 1mS para 6,144Mhz = 512 */    \
```

```
     TH0 = 0x02;     /* 1mS para 6,144Mhz = 512 */     \
}                                                       \
void _nsTimerPulseOn(    unsigned char low,       \
                              unsigned char high)       \
{                                                       \
     low--;                                             \
     _nsTimerPulseLow = low;                            \
     _nsTimerPulseLi = low;                             \
     _nsTimerPulseHigh = high;                          \
     _nsTimerPulseHi = high;                            \
     TMOD = TMOD & 0xF1;                                \
     TL0 = 0x00; /* 1mS para 6,144Mhz = 512 */    \
     TH0 = 0x02;     /* 1mS para 6,144Mhz = 512 */     \
     P1_BITS.B0 = 0;                                    \
     set_vector(TIMER0, _nsTimerPulseIsr);       \
     PT0=1; ET0=1; EA=1; TR0=1;                         \
}                                                       \
void _nsTimerPulseOff(void)                             \
{                                                       \
     PT0=0; ET0=0; TR0=0;                               \
}
#endif
```